P R O M A L

(PROgrammer's Micro Application Language)

THE PROMAL PROGRAMMING SYSTEM MANUAL

For The APPLE IIe and IIc
-or-
COMMODORE 64

PART 1: MEET PROMAL!
PART 2: PROMAL USER'S GUIDE
PART 3: PROMAL LANGUAGE
PART 4: PROMAL LIBRARY

SYSTEMS MANAGEMENT ASSOCIATES, INC.
3325 Executive Drive
Raleigh, North Carolina 27609
(919) 878-3600

Rev. C - September 1986

ACKNOWLEDGEMENTS

The PROMAL Language, Compiler, Executive,
and Library were designed and written by:

Bruce D. Carbrey

The PROMAL Editor and portions of the Apple IIe/IIc and
Commodore 64 I/O Interface were written by:

Larry Isaacs

with programming assistance from

David Long

TABLE OF CONTENTS

PART 1: MEET PROMAL!

Hardware Requirements, Copyright Notice, Trademarks..............1-2
Notice from Apple Computer Inc.....................................1-2
Meet PROMAL!...1-3
About the Promal Demo Diskette.....................................1-3
Audience...1-3
About this Manual..1-3
What is PROMAL?..1-3
Duplicating the Demo Diskette......................................1-4
Loading PROMAL...1-5
Using the PROMAL Executive...1-6
Writing a Program with the Editor..................................1-9
Compiling your first Program.......................................1-11
Executing your Program...1-12
Where does PROMAL put the Program?.................................1-12
Revising your Program..1-14
A Sample PROMAL Text-processing Program (FIND)......................1-15
Real (floating point) Numbers & Simple Business Program............1-24
An Advanced Program (CALC)...1-24
Advanced Editor Features...1-25
Some Special Capabilities..1-29
A Database Application Program (Apple II)..........................1-30
Sprites, Animation & Sound Synthesis (Commodore 64)................1-30
In Conclusion..1-32
Developer´s Version, Source Code, & Graphics Toolbox...............1-33
Customer Service...1-34

PART 2: PROMAL USER´S GUIDE

Introduction...2-2
Manuals & System Startup...2-3
Executive Commands and Command Editing.............................2-4
Built-in Executive Commands..2-8
Arguments for Executive Commands...................................2-9
File Names...2-9
File Name Extensions...2-12
Numeric Arguments..2-12
Device Names...2-13
I/O Redirection..2-14
PROMAL Executive Command Summary...................................2-14
BUFFERS (Apple II only)..2-15
COLOR..2-16
COPY (EXTCOPY)...2-17
CS...2-19
DATE...2-20
DELETE...2-20
DISKCMD (Commodore 64 only)..2-21
DYNO (Commodore 64 only)...2-22
DUMP...2-23
EDIT...2-24
FILES (EXTDIR)...2-24
FILL...2-26
FKEY...2-26
GET..2-27
GO...2-28

TABLE OF CONTENTS

HELP..........2-29
JOB..........2-30
LOCK (Apple II only)..........2-32
MACRO..........2-32
MAP..........2-33
NEWDIR (Apple II only)..........2-36
NOREAL..........2-36
PAUSE..........2-37
PREFIX (Apple II only)..........2-38
QUIT..........2-38
RENAME..........2-39
SET..........2-39
SIZE..........2-40
TYPE..........2-41
UNLOAD..........2-42
UNLOCK (Apple II only)..........2-42
WS..........2-43

The PROMAL Editor..........2-44
Display Screen for New File..........2-45
Editing Keys..........2-47
Inserting and Deleting Lines..........2-49
Searching with the FIND Key..........2-50
Search and Replace..........2-51
Cut and Paste Operations..........2-52
Saving a Block to a File..........2-52
Exiting the Editor..........2-53
Technical Notes on the Editor (C-64 only)..........2-47

The PROMAL Compiler..........2-56
Command line Arguments and Options..........2-56
Dialogue of the Compilation Process..........2-58
Differences between Demo and Full Compiler..........2-60
PROMAL Cross-Reference Map Utility (XREF)..........2-60

PART 3: PROMAL LANGUAGE MANUAL

Chapter 1: Introduction..........3-2
Chapter 2: PROMAL Programming Language Overview..........3-3
Chapter 3: Elements of the PROMAL Language..........3-7
Vocabulary and Names..........3-7
Data Types..........3-8
Literal Numbers, Characters, and Strings..........3-9
Variables..........3-12
Constant Definition..........3-13
Array Variables..........3-14
DATA Definition..........3-15
Operators and Arithmetic Expressions..........3-17
Relational Operators..........3-21
Logical Operators..........3-21
Shift Operators..........3-22
Indirect and Address Operators..........3-22
Global Variables..........3-24
Chapter 4: Statements..........3-25
Introduction..........3-25
PROGRAM Statement (& OVERLAY Statement)..........3-25
Assignment Statement..........3-26

TABLE OF CONTENTS

Conditional Statements.......................................3-26
IF...3-26
WHILE..3-28
REPEAT...3-29
FOR..3-29
CHOOSE...3-30
BREAK..3-31
NEXT...3-32
NOTHING..3-32
ESCAPE and REFUGE..3-33
Chapter 5: Procedures and Functions......................3-36
Built-in Procedures and Functions............................3-36
Simple Output & Fomatted Numeric Output......................3-37
Simple Input...3-38
Numeric Input..3-39
User-Defined Subroutines.....................................3-41
Passed Arguments...3-42
Local Variables..3-45
OWN Variables..3-46
Good Programming Practices with Subroutines..................3-46
Recursion..3-47
INCLUDE Statement for Multiple Source Files..................3-48
LIST Statement for Selective Listing.........................3-48
Conditional Compilation......................................3-49
Chapter 6: Interfacing...................................3-51
Opening and Closing Files and Devices........................3-51
Functions for File and Device I/O............................3-52
STDIN and STDOUT File Handles................................3-53
Output to Printer, Control and ESCape Sequences..............3-54
Argument Passing from the Executive..........................3-56
EXTernal Variables for Special Memory Locations..............3-57
PROMAL Interface to Machine Language.........................3-60
Chapter 7: Strings and Arrays Revisited..................3-61
Strings..3-61
Address Versus Content of an Array Element...................3-62
Multidimensional Arrays & Arrays of Strings..................3-64
Presetting Global Variables..................................3-66
Chapter 8: The Loader....................................3-67
Definitions..3-67
Modules & How the Loader Works...............................3-68
Calling the Loader and Options...............................3-70
Exporting & Importing Variables and Subroutines..............3-73
Executing the Logical Program with Separate Modules..........3-75
Using a Bootstrap to Control Loading.........................3-76
Using Overlays...3-77
Considerations for the Executive and Editor..................3-79
Successful Use of Overlays & Separate Compilation............3-81

PART 4: PROMAL LIBRARY MANUAL
Summary of Library Routines...................................4-2
How to use the Library Routine Descriptions...................4-4
PROC ABORT, FUNC ABS..4-5
FUNC ALPHA, PROC BLKMOV.......................................4-6
FUNC CHKSUM, PROC CLOSE.......................................4-7
FUNC CMPSTR...4-8
FUNC CURCOL, FUNC CURLINE.....................................4-9
PROC CURSET, FUNC DIR...4-10

TABLE OF CONTENTS

FUNC DIROPEN (Apple II only)....................4-11
FUNC EDLINE....................4-12
PROC EXIT, PROC FILL....................4-14
PROC FKEYGET....................4-15
PROC FKEYSET....................4-16
FUNC GETARGS....................4-17
FUNC GETBLKF....................4-18
FUNC GETC....................4-19
FUNC GETCF, FUNC GETKEY....................4-20
PROC GETL....................4-21
FUNC GETLF....................4-22
FUNC GETPOSF (Apple II only)....................4-23
FUNC GETTST, FUNC GETVER....................4-24
FUNC INLINE, FUNC INLIST....................4-25
FUNC INSET....................4-27
PROC INTSTR....................4-28
PROC JSR, FUNC LENSTR....................4-29
PROC LOAD, FUNC LOOKSTR....................4-30
FUNC MAX....................4-31
FUNC MIN, FUNC MLGET....................4-32
PROC MOVSTR....................4-33
FUNC NUMERIC....................4-34
FUNC ONLINE (Apple II only)....................4-35
FUNC OPEN....................4-36
PROC OUTPUT....................4-41
PROC OUTPUTF....................4-43
PROC PROQUIT, PROC PUT....................4-44
PROC PUTBLKF....................4-45
PROC PUTF, FUNC RANDOM....................4-47
PROC REALSTR....................4-48
PROC REDIRECT....................4-49
FUNC RENAME....................4-50
PROC SETPOSF (Apple II only)....................4-51
FUNC SETPREFIX (Apple II only), FUNC STRREAL....................4-52
FUNC STRVAL....................4-53
FUNC SUBSTR, FUNC TESTKEY....................4-55
FUNC TOUPPER....................4-56
PROC WORDSTR, FUNC ZAPFILE....................4-57

APPENDICES

APPENDIX A: ASCII CHARACTER SET....................A-1
APPENDIX B: PROMAL KEY CODES RETURNED BY FUNCS GETKEY AND TESTKEY....B-1
APPENDIX C: ERROR MESSAGES & MEANINGS....................C-1
APPENDIX D: LOCATING RUNTIME ERRORS & VARIABLES IN MEMORY............D-1
APPENDIX E: PRINTER SUPPORT....................E-1
APPENDIX F: DATA COMMUNICATIONS SUPPORT....................F-1
APPENDIX G: PROMAL MEMORY MAPS....................G-1
APPENDIX H: DYNAMIC MEMORY ALLOCATION....................H-1
APPENDIX I: MACHINE LANGUAGE INTERFACING....................I-1
APPENDIX J: RECURSION AND FORWARD REFERENCES....................J-1
APPENDIX K: REAL FUNCTION SUPPORT....................K-1
APPENDIX L: COMPATIBILITY ISSUES....................L-1
APPENDIX M: COMMODORE 64 RELATIVE FILE SUPPORT....................M-1
APPENDIX N: OPERATING SYSTEM NOTES....................N-1
APPENDIX O: BACKING UP & FORMATTING DISKETTES....................O-1
APPENDIX P: SYNTAX DIAGRAMS FOR THE PROMAL LANGUAGE....................P-1
APPENDIX Q: PROMAL DEMO PROGRAMS....................Q-1

M E E T P R O M A L !

(**PRO**grammer's **M**icro **A**pplication **L**anguage)

AN INTRODUCTION TO THE PROMAL PROGRAMMING SYSTEM

For APPLE IIe, IIc and COMMODORE 64 Computers

SYSTEMS MANAGEMENT ASSOCIATES, INC.
3325 Executive Drive
Raleigh NC 27609

Rev. C - Sep. 1986

MINIMUM HARDWARE REQUIREMENTS

Apple II: IIe or enhanced IIe, with Extended 80 column card (128K) and one floppy disk, or Apple IIc. PROMAL supports ProDOS, not DOS 3.3. Sorry, PROMAL does not work on an Apple II **Plus**. PROMAL works with (but does not require) RAMWORKS or equivalent memory expansion for /RAM disk.

Commodore 64 or 128: One 1541/1571 or true compatible disk drive, or MSD drive. Note: Most "Turbo" or other fast disk load accessories are incompatible with PROMAL, and should be removed before running PROMAL. However, PROMAL has DYNODISK built-in, which doubles the reading speed of 1541 or 1571 disk drives. This feature can be disabled if desired. PROMAL is compatible with the Skyles 1541 Flash and IEEE Flash.

COPYRIGHT NOTICE

TRADEMARKS

DISCLAIMER

Systems Management Associates, Inc. makes no claims or warranties with respect to the contents of this publication or the product it describes, beyond the limited warranty included with the product. Further, the right is reserved to make any changes to this publication or the product it describes without obligation to notify any person of such revision or changes.

NOTICE FROM APPLE COMPUTER INC. FOR APPLE VERSION

ProDOS is a copyrighted program of Apple Computer, Inc., licensed to Systems Management Associates, Inc. to distribute for use only in combination with PROMAL. Apple Software shall not be copied onto another diskette (except for archival purposes) or into memory unless as part of the execution of PROMAL. When PROMAL has completed execution Apple Software shall not be used by any other program.

Apple Computer Inc. makes no warranties, either express or implied, regarding the enclosed software package, its merchantability or its fitness for any particular purpose. The exclusion of implied warranties is not permitted in some states. The above exclusion may not apply to you. This warranty provides you with specific legal rights. There may be other rights that you may have which vary from state to state.

MEET PROMAL!

Congratulations! You're about to meet PROMAL, the most powerful integrated programming environment you can buy for your computer. This manual can be used in conjunction with the PROMAL Demo Diskette to help you get acquainted with PROMAL.

About the PROMAL Demo Diskette... Don't let the name "Demo" fool you. This disk contains a fully functional PROMAL system. It contains all of the features of the sealed System Disk, **except the ability to compile large programs** (400 lines, maximum, excluding comments). Before opening your sealed System Diskette you should use the Demo Diskette for evaluation. **PROMAL IS NOT COPY-PROTECTED AND ONCE THE SEAL IS BROKEN ON THE SYSTEM DISKETTE IT CANNOT BE RETURNED FOR A REFUND.** All the files referred to in this MEET PROMAL manual will be found on the PROMAL Demo Diskette. This way you can satisfy yourself that PROMAL is everything we say it is. We want you to be 100 percent satisfied with our product.

AUDIENCE

We do not claim that PROMAL is for everyone. PROMAL is for the person who has a working familiarity with the computer and BASIC (or another high-level language), and wants to get the most from his or her computer. If you are frustrated by the limitations of BASIC but want to avoid the complexity and low productivity of machine-language programming, then PROMAL is for you. If you want to realize the full potential of your personal computer, you need the power, the speed, and the capabilities of PROMAL.

ABOUT THIS MANUAL

Your PROMAL system includes four manuals plus Appendices and an Index. The first manual, "MEET PROMAL!", which you are reading, will not make you a proficient PROMAL programmer, but will provide enough information for you to evaluate the PROMAL system. The "MEET PROMAL!" Manual is required reading for the owner of the PROMAL System; please do not attempt to study the USER'S GUIDE or LANGUAGE MANUAL until you have read and executed the examples in "MEET PROMAL!". This approach will help even the advanced programmer get the most from PROMAL.

WHAT IS PROMAL?

PROMAL (PROgrammer's Micro Application Language) is a high-performance programming system. It includes:

* An Interactive Command EXECUTIVE
* A Full-Screen EDITOR
* A Structured Language COMPILER
* An on-line LIBRARY of predefined subroutines

The **EXECUTIVE** lets you enter commands to load and execute programs, to manage files and memory, to display files and directories, and to do much more. Many built-in commands are provided, and you can add as many of your own commands as you want.

The full-screen **EDITOR** makes it easy for you to create or modify PROMAL programs or other text files. It has powerful features similar to many word-processors, but is designed specifically to simplify program generation.

The **COMPILER** converts your program into a very compact, very fast-executing command (or "object program") which you can run by just typing its name. Compiled PROMAL programs will often run 20 to 100 (or more!) times faster than BASIC, and occupy much less memory.

The **LIBRARY** provides an extensive base of built-in, pre-programmed subroutines which are always available to your programs. The library greatly simplifies programming by providing a nucleus of commonly-needed functions.

ARE PROMAL PROGRAMS LIKE BASIC PROGRAMS?

Although your previous experience with BASIC will be helpful in understanding PROMAL, you will find very much that is different. Some aspects of PROMAL may seem strange at first, but you will soon appreciate its simple elegance. Once you have mastered a few basic concepts, you will find PROMAL much easier to program than BASIC for any non-trivial application.

LET'S GET STARTED!

If you're ready to begin your guided tour, get your PROMAL Demo Diskette, your computer, and read on ...

DUPLICATING THE DEMO DISK

Because the Demo Diskette contains important files which are **not** duplicated on the sealed System Diskette, <u>you should make a copy of the Demo Diskette before proceeding</u>. The examples in MEET PROMAL will write on the demo disk. Therefore you should <u>only use the copy</u>, so that if you make a mistake or accidentally overwrite or delete a file, you will still have the original Demo Disk. You can copy the Demo disk with any kind of disk copy program. Commodore owners who don't have a copy program can use the DISKETTE utility, as described in **Appendix O.** Apple users can use the standard ProDOS disk copier furnished on the System Utilities Disk which came with your computer.

LOADING PROMAL

<u>For Apple II:</u>

Put your **copy** of the PROMAL <u>Demo</u> Diskette into the disk drive, close the door, and turn on your computer. If your computer is already on, you can hold down the CTRL and open-Apple keys and press RESET.

If you have a hard disk and wish to boot from hard disk, create a new directory, and copy all the files from the Demo disk onto it. The volume name for the Demo Disk is /PROMAL/. Then execute PROMAL.SYSTEM.

If you have a Ramworks II or similar /RAM disk and want to use it with PROMAL, first install your /RAM device as usual. Then execute PROMAL.SYSTEM. This can be done from BASIC with the command **-/PROMAL/PROMAL.SYSTEM.**

Please note that **only the Demo disk** contains a bootable version of PROMAL. The System disk (and optional Developer's disk) in the sealed envelope are crammed full of additional sample programs, utilities, etc., so there isn't enough room for duplicate copies of ProDOS, PROMAL.SYSTEM, etc. on each disk. Always boot up with a copy of the Demo disk; then you can change disks (using the **PREFIX *** command described later) to access files on the other disk(s).

For Commodore 64/128:

Please follow the following steps **carefully.**

1. Turn on your computer system in the usual manner.

2. If you have a Commodore **128,** you must type **GO 64,** and reply **Y** to the "Are you sure?" prompt. PROMAL works only in Commodore 64 mode. Do not attempt to use 128 mode or non-1541 compatible disk features of the 1571 disk (such as double-sided mode or "fast" mode).

3. Insert your copy of the Demo Disk in the drive, close the door and type:

LOAD "PROMAL",8 <RETURN>

4. If the **only** peripheral on your system is a single 1541 or 1571 disk drive, proceed to step 5. The Commodore 64 version of PROMAL includes DYNODISK, which doubles the reading speed of the Commodore 1541 or 1571 disk drive. This feature is comparable to many other disk speedup products which you may have or may have seen. PROMAL is compatible with Skyles Flash products provided you disable DYNODISK as described below. Do not use PROMAL with any other disk speedup cartridge or software package unless you are **sure** it does not use **any** memory in the Commodore 64 and is completely 1541-compatible; otherwise, PROMAL will not work properly. DYNODISK works automatically with all PROMAL programs; there is nothing to install. DYNODISK has the same limitations as other disk speedup products, namely:

(a) You can't use DYNODISK if you have any other device (such as a printer or second disk drive) attached to the computer on the serial bus. Therefore if you have a printer, you should either turn it off while reading from disk, or disable DYNODISK as described below. If you decide to leave DYNODISK enabled and turn off your printer, and you have a printer interface (such as a CARDCO), you will need to turn it off too (in the case of the CARDCO G+, this means disconnecting the single wire from the back of the computer). Failure to observe this precaution may cause the computer to "hang up", necessitating reloading PROMAL. DYNODISK can be re-enabled by a simple command when PROMAL is running.

(b) If you don't have a 1541 or 1571 disk drive, see **APPENDIX N** before continuing.

If you need to **disable** DYNODISK, type:

POKE 3555,0 <RETURN>

5. Type:

RUN <RETURN>

For Either computer:

The system will then load the EDITOR, EXECUTIVE, and LIBRARY. On the screen you will see:

```
LOADING EDITOR
LOADING EXECUTIVE
LOADING LIBRARY...
```

The screen will then clear and display:

```
PROMAL Development System EXECUTIVE
            Version 2.1
    Copyright (C) 1986 SMA Inc.

     PROMAL is a Trademark of
 Systems Management Associates, Inc.
```

At the end of the signon information, you will see (unless you have a Thunderclock card in an Apple II system, which sets the date automatically):

```
PLEASE ENTER TODAY'S DATE
(in the form MM/DD/YY):
```

Type in the correct date, for example 09/30/86, and press RETURN. PROMAL uses this date to automatically "tag" all compiled programs with their creation date, so if you make revisions to a program, you can easily see which is the current version. You will now see:

Apple II	Commodore 64
F1 = EDIT	F1 = EDIT
F2 = PREFIX *	F2 = DUMP
F3 = COMPILE	F3 = COMPILE
F4 = GET	F4 = GET
F5 = FILES	F5 = FILES
F6 = EXTDIR *	F6 = MAP
F7 = HELP	F7 = HELP
F8 = COPY	F8 = COPY

These lines tell what the default function keys do. On the Apple II, the letter "F" may be replaced by an "Apple" icon. **On the Apple II, a function key is activated by holding down either Apple key and pressing the desired number.** You may use function keys to quickly enter commands when using the EXECUTIVE. Now lets begin our guided tour of the PROMAL EXECUTIVE.

USING THE PROMAL EXECUTIVE

The "-->" is the PROMAL EXECUTIVE's prompt, which indicates that the EXECUTIVE is ready for you to type in a command. Try typing this:

TYPE README.T

followed by <RETURN>. All commands need to be terminated by the <RETURN> key. This command will display additional information about your system not included in this manual. If instead you get a "FILE NOT FOUND" error, just go on.

Now try typing:

hello

followed of course by <RETURN>. You should see:

```
--> hello

PROGRAM OR OVERLAY NOT FOUND: HELLO

--> _
```

Here's what happened. When you typed "hello", PROMAL tried to execute the command called "hello". Since there is no built-in command called "hello", the EXECUTIVE searched for a user-defined command file on disk. Since it didn't find the command file "hello", it gave you an error, and is now ready for another command. This is how PROMAL executes commands. First it looks for the command in memory, and then on disk. (You can create new commands by compiling a PROMAL program.) Unlike BASIC, you can have several programs in memory at once, and execute any of them by just typing the name. In fact, the EXECUTIVE and EDITOR are just compiled PROMAL programs that are already in memory!

Now press function key **F5** (for Apple II, hold down either Apple key and press 5). You will see "FILES" appear. This is a built-in command. Press <RETURN> and the EXECUTIVE will execute the FILES command. Try it. You should see a list of file names. On the Apple these names will be displayed in four columns. On the Commodore, the display will be similar to a directory listing made from BASIC.

These are the file names on your demo diskette. Notice that most of the filenames end in ".C" or ".S". This is because PROMAL file names have a single character file extension after the name which tells what **kind** of file it is. ".C" files are compiled command programs and ".S" files are PROMAL language source (text) programs which can be edited. Therefore FIND.S is a source program and FIND.C is the compiled (executable) form of the same program.

For the Apple II only, you can get more information about the files on the disk by using the EXTDIR command (EXTended DIRectory). Press F6 (Apple key with 6), **or** type **EXTDIR** * and press <RETURN>. Be sure to type a space before the asterisk. You should see a list of all the files, with information about each file, and a summary showing how many blocks are free on disk. Unlike the FILES command, the EXTDIR command is not built-in to the EXECUTIVE. Instead, EXTDIR is actually a PROMAL program which is loaded from disk and executed when you type EXTDIR.

IMPORTANT: For the Apple II only, please note that if you **change disks** you will need to press **F2** and <RETURN> after changing disks to issue a **PREFIX *** command. This tells ProDOS what the new volume name is; otherwise, it will still look for your old diskette, probably generating DEVICE NOT READY or FILE NOT FOUND errors.

Now press **F7** and <RETURN> (or just type in **HELP** - the result is the same). You will see the screen scroll up to display a "help" page similar to:

```
                PROMAL HELP
CTRL-                       CTRL-
E Enter insert mode         D Delete char.
B Recall Prior Line         F Cursor to first
\ Clear To End Line         L Cursor to last
       PARTIAL COMMAND SUMMARY
COMPILE [File [L[=List]][O=Object]
COPY File [Dest.][#Drvs]
DELETE File                Function Keys
DUMP From [To]              f1 = EDIT
EDIT [File]                 f2 = PREFIX *
FILES [Dir]                 f3 = COMPILE
FILL From To Value          f4 = GET
FKEY [Number String]        f5 = FILES
GET Commandfile             f6 = EXTDIR *
JOB File.J [Arg...]         f7 = HELP
MAP                         f8 = COPY
PREFIX [/Path/ or *]
RENAME File Newname
SET Addr Val [Val...]
TYPE File
UNLOAD [Command]
--> _
```

The display above is for the Apple II; the Commodore display will differ somewhat. The top four lines are clues to using some control keys for line-editing while you are in the EXECUTIVE. We'll get to them in a minute. The rest of the lines summarize many of the most commonly-needed commands. Many commands need or may have **arguments** after the command name. These arguments are usually file names or numbers. Arguments shown in square brackets ([]) on the help screen are optional. For example, DUMP needs one argument, "From", and can optionally have a second argument, "To". Try typing this:

```
dump 1100
```

You will see a display similar to:

```
1100  72 65 7A 85 72 A5 71 65   rez.r.qe
```

The actual numbers and letters shown will be different. This is the contents of memory locations 1100 hex through 1107, displayed in hex and ASCII. A "." is shown for any byte which doesn't represent a printable ASCII character. If you're not familiar with hex notation, don't be concerned; later you may want to consult the reference manual for your computer to learn about hexadecimal. DUMP with one argument will display 8 bytes starting at the FROM address.

Now suppose you wanted to dump from 1100 to 1180. Wait! If you think you know, don't type yet! Instead press **CTRL**, and while holding it down, press **B**. What happened? You should see:

```
--> dump 1100_
```

You can recall your last command by using CTRL-B! Now you can "edit" your prior command by just typing " 1180" at the end of the line. Don't forget the space between 1100 and 1180; PROMAL expects a space between arguments. Press <RETURN>. You should see a display of memory from 1100 to 1180.

You will find CTRL-B very useful for correcting mistakes in commands. You can recall the prior command and edit it again, rather than re-typing all of it. There are several keys you can use to edit command lines, but we'll save these until we discuss the EDITOR (unless you want to experiment right now using the clues given by HELP).

One more thing about CTRL-B. Try pressing it several times. What happens? You can "backtrack" through prior commands, one at a time! After you get all the way back to the start of the session (before you typed in the date), the next CTRL-B will "wrap around" again to the most recent command.

WRITING A PROGRAM WITH THE EDITOR

Now that you know a little about using the EXECUTIVE, let's try writing a program using the EDITOR! Press **F1** and <RETURN> (or type **EDIT**). Almost instantly the screen will clear, except for the bottom 5 lines which show:

```
--------------------------------------------------------------------------

LINE =     1
1=DEL LN  2=INS LN  3=MARK   4=RECALL  5=FIND   6=CHANGE  7=HELP   8=QUIT
```

The top 20 blank lines form your text area, and the bottom 5 lines are a status area, which tells you that the function keys have now been redefined. On the Commodore 64, these function key legends occupy two 40-column lines instead of one as shown above. For example, F1 now means "delete line". You won't need any of these function keys (except F8=QUIT) to edit your first PROMAL program.

The blinking cursor is at the top of the screen. It always indicates where the next character you type will appear. Try typing in the following program, exactly as shown below. Be sure to start each line in the first column, and use <RETURN> to end each line.

```
PROGRAM HELLO
INCLUDE LIBRARY
BEGIN
PUT "HELLO YOURSELF!",NL
END
```

If you make a mistake, use the DELETE key to delete characters. You can use the cursor keys to backup and over-type any corrections too. You can type in upper or lower case or a mix - it doesn't matter. If you prefer upper case, you may want to press CTRL-A, which will cause ALPHALOCK to be displayed in the status area. After this, any alphabetic keys you type will automatically be in upper case.

One thing you probably noticed right away is that this program doesn't have any line numbers. PROMAL programs don't need them. You'll see later how you make branches and subroutine calls without line numbers. Another thing that's different about PROMAL programs is that you can only put one statement on a line. This makes programs more readable and easier to change. If you think about it, the main reason you put multiple statements on one line in a BASIC program was to avoid using another line number. Since PROMAL has no line numbers, there's no need for several statements on one line.

The first line of your program simply gives it a name (HELLO). Every PROMAL program starts this way. The next line tells PROMAL to include all the subroutines in the built in LIBRARY. You will normally use the INCLUDE LIBRARY statement in every program you write. The "BEGIN" line merely indicates the start of the main program. The next line is the substance of your little program. PUT is similar to a BASIC PRINT, and prints strings or characters. Unlike BASIC, you have to tell PROMAL explicitly when you want to start a new line. The **NL** does this (it stands for New Line). You can also use **CR** (Carriage Return) in place of NL; the result is the same.

When you're done press **F8**. You will see a display similar to that below. The numbers on your screen may be different (especially on a Commodore 64). This display shows your choices of what to do. "WORKSPACE", is a small, in-memory file with the name "W". It is useful for temporarily storing a small source program you are experimenting with, without saving it on disk.

Press W and <RETURN>. Immediately, the cursor will re-appear below the "SELECTION?" prompt. This indicates that your text has been written to the workspace (it's fast!), and you may now make another selection.

```
        PROMAL EDITOR 2.1
    Copyright (C) 1986 SMA, Inc.

 BUFFER SIZE = 9862
 FILE NAME = W (AUTO UPDATE ON QUIT)
 FILE SIZE = 66

              OPTIONS

     R = REPLACE ORIGINAL FILE
     N = WRITE TO NEW FILE
     W = WRITE TO WORKSPACE
     C = CONTINUE EDITING
     Q = QUIT EDITOR

  SELECTION?

  _
```

Type **Q** and <RETURN>, and the EXECUTIVE prompt reappears:

```
-->  _
```

COMPILING YOUR FIRST PROGRAM

Unlike BASIC, you must COMPILE your program before you can run it. Compiling it does not destroy your source (text) program, but generates a separate ("object") program which is executable.

Press **F3**, (or type **compile**) and press <RETURN>. After a pause while the COMPILER loads from disk, you will see it "sign on", followed by a rapidly changing display as your program compiles. This will only take a few seconds. The compiler will display READING LINE XX, where the XX's change rapidly to show you what line number it's working on. The final display should look similar to this.

```
      PROMAL DEMO COMPILER 2.1
     Copyright (C) 1986 SMA Inc.

READING LINE 2
INCLUDING LIBRARY
READING LINE 67
RESUMING FILE W
READING LINE 68
COMPILING <MAIN PROGRAM>
READING LINE 70
HELLO compiled:
  70 Source lines
  $0025 (37) bytes Obj.
  $0 Scalar, $0000 (0) tot. Vars.
Table usage:
  Symbols: 36%  Fwd. Refs: 1%
  Strings: 1%

--> _
```

The display may differ slightly, especially for the Commodore 64. The summary at the end shows information about the **object file** which the compiler wrote to disk. This object file is your executable program, with the default name HELLO.C.

You may be wondering why the compiler said it compiled 70 source lines (less for the Commodore 64). Your program was only 5 lines long! The answer is that the INCLUDE LIBRARY line of the program caused the COMPILER to read another file called the LIBRARY at that point in your program. Later when you start writing big programs, you can tell it to include other files of your own. For example, you can tell it to INCLUDE some file of subroutines. In this way, you can share commonly-used routines between many programs without having to re-type them or "paste" them into your program. The LIBRARY file contains about 63 lines of standard definitions (59 for the Commodore) that you will want in nearly every PROMAL program. Like the Workspace, the LIBRARY is a memory-resident file.

The summary also tells how big your compiled program is (37 bytes), how many bytes are needed for its variables (none in this case), and some information about how much of the compiler's internal tables were used. This is described in the PROMAL USER'S MANUAL. Don't be concerned that your little

program took up 36 percent of the symbol table; the demo compiler has a small symbol table. The full compiler can compile programs with thousands of lines.

EXECUTING YOUR PROGRAM

Now that the compiler is done, your program is ready to execute. Just type:

hello

And your program will display:

```
HELLO YOURSELF!

--> _
```

When your program finished, it returned control to the EXECUTIVE. You have now created a new, user-defined command for the EXECUTIVE! Your **object** program is saved on disk as file HELLO.C, and will be run anytime you type HELLO from the EXECUTIVE. You can save as many commands as you want on as many diskettes as you like. To execute a command, just type its name. If it's been executed before and is in memory, it will execute instantly. If not, the EXECUTIVE will fetch it from disk and execute it.

Note that you have not yet saved your **source** program (the text file you can edit) on disk; it's only in the temporary workspace. Normally, you will want to save your source program on disk when you exit from the EDITor. If you want to, you can still save it now, from the EXECUTIVE, by typing **COPY W HELLO.S** which copies the Workspace to a new disk file called HELLO.S.

WHERE DOES PROMAL PUT THE PROGRAM?

You might be curious to know where your program is in memory. Type:

MAP

and you will see a "map" of memory, which tells where PROMAL put your program and how the remaining memory is allocated. The top half shows information about what program(s) you have in memory, and the bottom half shows a summary of available space. We won't go into all the details of the MAP display here. It is fully explained in the PROMAL USERS GUIDE.

<u>Apple II</u>

```
HELLO        (PRO.) 09/30/86 CHKSUM 49CB
  CODE $2900-29FF

OBJECT PROGRAMS  $2900-29FF (256)
FREE SPACE       $2A00-8DFF (25600)
SHARED VARIABLES $8E00-     (0)
EXEC./EDIT SPACE $6100-8DFF (11520)
TOTAL SPACE      $2900-8DFF (25856)

ACTIVE WORKSPACE $1200-1241 (66)
FREE WORKSPACE   $1242-5AFF (18622)
```

Commodore 64

```
COMPILE     (PRO.) 11/ 4/85 CHKSUM 1465
  CODE $4F00-81FF, VARIABLES $8200-84FF
HELLO       (PRO.) 09/30/86 CHKSUM 4A86
  CODE $8500-85FF

OBJECT PROGRAMS   $4F00-85FF (14080)
FREE SPACE        $8600-98FF (4864)
ACTIVE WORKSPACE  $9900-9941 (66)
FREE WORKSPACE    $9942-A0FF (1982)
SHARED VARIABLES  $A100      (0)
EXEC./EDIT SPACE  $A200-CFFF (11776)
TOTAL SPACE       $4F00-CFFF (33024)
```

For the Apple version, you may have two programs in memory (EXTDIR and HELLO), or just one if you didn't try EXTDIR before. The Commodore version shows two programs in memory, COMPILE and your HELLO program. Since the Apple has somewhat less memory than the Commodore, but has a relatively fast disk, the Compiler is always unloaded automatically when it finishes, leaving more room for other programs. Since the Commodore has a little more available memory but a slow disk, PROMAL normally keeps the compiler in memory when it finishes so you won't have to wait for it to load when you need it again. The number after "CODE" shows the address where your program was loaded.

If you executed another program, a part of PROMAL called the **loader** would put it right above where HELLO ends. PROMAL always allocates programs and data on exact "page" boundaries in memory (that is, the starting hexadecimal address will always end in 00). Technically, PROMAL programs are known as "relocatable" object code, which means that they can be run **anywhere** in memory. In the event that you use up so much memory that there isn't enough room to load the specified command program, PROMAL will automatically "unload" programs to make enough room. The MAP also tells other information about your HELLO program. It tells you it is a PROMAL program, that it was compiled on 11/04/85 (or whatever date you used), and what its "checksum" is. If your program uses any variables (your HELLO program doesn't), it shows what space is allocated for variables.

The checksum requires more explanation. When PROMAL loads a program into memory, it computes a 16-bit sum of all the bytes loaded. This is the checksum shown, in hex. Anytime you execute that program, PROMAL recomputes the checksum and compares it to the saved value. If the two values don't match, the loader knows something has corrupted the program in memory (such as another program POKEing around where it shouldn't!). The loader then automatically reloads the program from disk. This provides an integrity check for your programs.

The meanings of the rest of the summary are described in the PROMAL USER GUIDE. One thing you might be interested in now though is the line labelled TOTAL SPACE. This is the maximum amount of memory that you can use for PROMAL programs; about 25K bytes for the Apple or 33K bytes for the Commodore. Although this amount is somewhat less than is allowed by BASIC, you can still have PROMAL programs that are **much** larger and more complex than is possible using BASIC, because PROMAL programs are much more compact. For example, the

COMPILEr itself is a PROMAL program of over 4000 lines, yet occupies only about 14K bytes. In addition, PROMAL allows you to use **overlays** to run programs that are larger than memory. This is described in the PROMAL LANGUAGE MANUAL.

You won't concern yourself often with how PROMAL allocates memory, because it is automatic. We have examined it briefly here to give you an idea of how the PROMAL program loader really works, and to touch on some of the technical concepts behind the power of the PROMAL System.

REVISING YOUR PROGRAM

Let's try making a small change to your program. Re-enter the EDITOR by pressing **F1** and <RETURN>. Your program will reappear in the text area of the display automatically.

Use the cursor keys to position the cursor over the "Y" in "YOURSELF" in the fourth line. Now press **CTRL-D** if you have an Apple II, or **CTRL-backarrow** (the key to the left of the "1" key) if you have a Commodore 64. What happened? This key deletes a character, but unlike the DELETE key, it pulls all the text on the line to the right over to "fill in the hole". Press this key 8 more times to get rid of "YOURSELF!". The line should look like this:

```
PUT "HELLO ",NL
```

Now put the cursor over the N in "NL" and press **CTRL-E** if you have an Apple or **SHIFT-INST** if you have a Commodore. The highlighted word INSERT appears in the status area at the bottom of the screen, indicating you are now in insert mode. In insert mode, anything you type will "push over" the text to the right of the cursor. With insert mode on, type:

CARG[1],

from your cursor position. To exit from insert mode, press any cursor key or <RETURN>. Your line should now look like this:

```
PUT "HELLO ",CARG[1],NL
```

In PROMAL, square brackets are used to enclose array subscripts instead of parentheses as in BASIC, so you can tell an array from a function easily. CARG is an array which is pre-defined (in the LIBRARY) for a special purpose. CARG is short for "Command Argument". The CARG array is an array of strings, which is automatically available to your program when it starts. CARG[1] is the first argument on the command line, CARG[2] is the second argument, etc.

Now press **F8** to exit, and select **W** and then **Q** to resave your new version to the workspace and exit to the EXECUTIVE. Now re-compile your program by pressing **F3** and <RETURN>. After your program compiles, type:

hello PROMAL

What happened? Your program displayed

```
HELLO PROMAL
```

Now try typing:

hello everybody

Whatever you type as the first argument gets passed to your program by the EXECUTIVE in the variable CARG[1], and your program prints it. Now try:

hello editor and executive

Your program will display:

HELLO EDITOR

What happened to "and executive"? Why didn't it display? Remember that a blank separates arguments on a command. Therefore "AND" got put in CARG[2], and "EXECUTIVE" got put in CARG[3], because the EXECUTIVE thinks they're each a separate argument. Your program only prints CARG[1].

You can force the EXECUTIVE to accept a string with blanks in it as a single argument by enclosing it in quotes. Try this:

hello "editor and executive"

You will now see:

HELLO editor and executive

If you were observant and if you were typing in lower case, you might have noticed another difference. Ordinarily the EXECUTIVE converts lower case commands and arguments to upper case as it reads them in. But if you enclose an argument in quotes, it won't convert the argument to upper case.

Naturally you can do more with command arguments than just print them. For example, you can use a command argument as a file name for your program to read or write. Our next example will demonstrate how to do this, and how you can do even more powerful "I/O redirection" on a command line.

Now that you've written a trivial PROMAL program, you may want to see a program that really does something useful. In the next section we'll take a look at a fairly short but useful program that illustrates a text-processing application, and illustrates many key PROMAL features, including using the LIBRARY.

If you want to postpone your exploration of these programs until later, you can simply turn off the computer.

A PROMAL TEXT-PROCESSING PROGRAM

Suppose you had a mailing list on a disk file called MAILLIST.T, which you had prepared with the PROMAL Editor. For simplicity's sake, suppose you had one entry per line, for example:

Board, Kim O., 6502 Processor St, Santa Clara, CA 95050

Suppose you had this file, and now you wanted to see all the customers in zip code 95050 of California. You could go buy a data base manager, right? Well, that's one way, but with PROMAL, a simple program may solve the problem. What you need is a command to find and display (or print) all the lines containing "CA 95050", for example:

```
FIND "CA 95050" MAILLIST.T
```

The FIND program to do this is already on your PROMAL Demo Diskette. To see the source program, type:

EDIT FIND.S

The EDITOR will "sign on", load the file into memory, and then display the first 20 lines of the program. The entire text of the program is reproduced on the following page for convenience.

After the first line, you will notice that there are a number of lines which start with a semicolon (;). These are comments. Any line which starts with ";" is a comment. Completely blank lines are considered comments, too. You can also put a ";" and a comment at the end of a statement. It is considered good programming practice to put enough comments in your program to adequately document it. It is important to realize that because PROMAL compiles your program, adding comments will not make your program use any more memory or execute slower. The compiler simply ignores all comment lines. This is an important difference from BASIC, where comments eat up valuable memory and increase execution time. Naturally adding comments to your source files will make them bigger, but this has no relation to the size of your executable object program. So feel free to comment!

Hold down the "**cursor down**" key and the EDITOR will "scroll" the text upward till you reach the end of the program. You can use the "cursor up" key to back up in the same way. Now scroll the screen until the line

```
INCLUDE LIBRARY
```

is on the top line of the display.

Now let's take a look at this program. Don't expect to understand all of it after this explanation; you just want to get the general idea of what a PROMAL program looks like and what some of the main concepts are. You'll need to study the PROMAL LANGUAGE MANUAL before you will actually be able to write or fully understand a complete program.

SOURCE PROGRAM "FIND.S"

```
PROGRAM FIND

;  by B. Carbrey 5/22/84
;  Program to print all lines in a text file which match a specified string.
;  Command syntax:  FIND <string> <file>
;  Examples:

;  FIND JANUARY MYDATA.T

;  will display all lines in the file MYDATA.T which contain the
;  word JANUARY.

;  FIND "New Jersey" MAILLIST.T > TEMP.T

;  will output all lines with New Jersey to the file TEMP.T from MAILLIST.T.
;  (quotes are needed if the string sought includes blanks).

INCLUDE LIBRARY

BYTE LINE[81]      ;Input/output buffer
WORD INFILE        ;Input file handle

FUNC BYTE HASSTRING  ; STRING
;  Returns true if LINE contains the desired STRING
ARG WORD STRING    ;desired string
WORD I             ;index to line
BEGIN
I=0
WHILE I < LENSTR(LINE)
  IF CMPSTR(STRING,"=",LINE+I,TRUE,LENSTR(STRING))
    RETURN TRUE
  I=I+1
RETURN FALSE
END

BEGIN ; main program...
IF NCARG <> 2  ; wrong # of arguments?
  PUT NL,"FIND error: 2 args. needed."
  PUT NL,"Usage: FIND string file"
  ABORT
INFILE=OPEN(CARG[2]) ; open data file
IF INFILE=0  ;open error?
  PUT NL,"FIND error: cant open ",CARG[2]
  ABORT
WHILE GETLF(INFILE,LINE) ; read a line
  IF HASSTRING(CARG[1])  ; string match?
    PUTF STDOUT,LINE,NL  ; yes, show
END  ;thats all folks!
```

First we need a few rules:

1. All variables must be defined (or "declared") before they are used.

2. Variables may have up to 31 characters. Unlike BASIC, which only looks at the first two characters, PROMAL uses all the characters to distinguish between variables. Also unlike BASIC, variables can contain PROMAL key words without causing trouble. Long variable names do not use any more memory than short variable names in your compiled program.

3. A variable definition tells the kind of data the variable represents. PROMAL supports the following data types:

BYTE: a single character, or an unsigned number from 0 to 255.
WORD: an unsigned number from 0 to 65,535, often used as an address.
INTEGER: a signed whole number between -32,767 and +32,767.
REAL: a "floating point" (decimal) number between -1.0E-37 and +1.0E+37.

4. Key words and variables must be separated from each other by blanks. They can't be run together as in BASIC.

5. Subroutines are given names (not line numbers), and can be either PROCedures or FUNCtions. A function starts with the key word FUNC, and returns a value to the calling program (much like BASIC). The function definition tells the type of data the function returns. Procedures start with PROC and do not return a value. They are similar to BASIC subroutines. Both procedures and functions are called by just putting the name in a statement. Functions and Procedures must be defined before they are called. In PROMAL, therefore, the main program always comes last, after all functions and subroutines.

With these rules in mind, let's look at few lines of the program:

```
BYTE LINE[81]           ;Input/output buffer
WORD INFILE             ;Input file handle
```

These two lines define two variables used by the program. The first variable is called LINE, and is of type BYTE. It is declared to have a dimension of 81. This variable will be used to hold a line of text read from the data file, up to 80 characters long. Note that PROMAL, unlike BASIC, does not have a primitive STRING data type. Instead, strings are treated as an array of bytes. This may seem like a shortcoming at first, but you will soon discover that PROMAL can actually manipulate strings very easily and much more efficiently than BASIC (experts in BASIC take note: this is because PROMAL never needs time-consuming "garbage collection"). By convention, a PROMAL string is an array of bytes terminated by a 0 byte (which is why LINE is dimensioned 81 to hold up to 80 characters).

INFILE is declared to be a variable of type WORD. Note that you still have to declare it, even though it is not an array. The comment says that INFILE is a "file handle". A file handle is a variable that is used to represent an active file. We'll demonstrate this later.

Now study the 13 lines beginning with:

```
FUNC BYTE HASSTRING   ; STRING
```

These 13 lines define a function called HASSTRING. The function ends with the END line. In BASIC you could only define functions with one argument, and only on one line. In PROMAL, a function can have any number of arguments passed to it and can have any number of lines. In our case, the line,

```
ARG WORD STRING   ;desired string
```

tells us that the function will expect one argument (ARG stands for "argument"), and it will be of type WORD and will be given the name STRING inside this function.

If BASIC is the only language you've ever used, this next part may be a little hard to grasp, so don't be concerned if you don't get it. Actually, the variable STRING will hold the **address** of whatever string the calling routine passes to it. If function HASSTRING is called like:

```
HASSTRING("CA 95050")
```

then the STRING variable will hold the address of a string of bytes in memory containing "CA 95050", terminated by a zero byte. This whole string can be manipulated using the STRING variable.

The line:

```
WORD I
```

declares a working variable for use within the function. Because this variable and STRING are both declared within the HASSTRING function, they are called **local** variables and only have meaning within the function. You may define other variables with the same names in other subroutines which would be completely different. This is an important concept. You may define and use variables in a subroutine without having to worry if you have already used the name for something different elsewhere. Variables declared before the first subroutine (such as LINE and INFILE) are **global** and may be used by all subroutines.

The BEGIN line starts the "action" part of the function. The next line:

```
I=0
```

will be the first statement executed when function HASSTRING is called. It sets I to 0. This is an assignment statement, just like BASIC. Note that variables are not automatically initialized to 0, as in BASIC, but contain "garbage" until you assign something to them.

Now let's take a look at the rest of the function definition:

```
WHILE I < LENSTR(LINE)
  IF CMPSTR(STRING,"=",LINE+I,TRUE,LENSTR(STRING))
    RETURN TRUE
  I=I+1
RETURN FALSE
END
```

There's a lot going on in these few lines! First of all, if you are looking at the lines on the 40 column Commodore display, one of the lines is only partially visible because it is longer than 40 characters. The last visible character is highlighted in reverse video so you can tell there is more to the line off-screen. To see the rest, just put the cursor on the line and move the cursor right. When it reaches the last column of the display, the whole line will scroll left to let you see the rest. Just press RETURN to restore the line to its normal position (Note: it is also possible to move the whole "window" to the right to view long lines - this is described in the PROMAL USER'S GUIDE).

The WHILE statement is one of several kinds of loops in PROMAL. It has no direct counterpart in BASIC, but is something like a combination IF and FOR-NEXT loop. A WHILE loop has the form:

```
WHILE condition
  statement 1
  statement 2
  ...
next statement
```

A WHILE statement tests the condition (like a BASIC IF statement). If the condition is TRUE, then all the indented statements (statement 1, statement 2, ...) are executed. After the last indented statement (...) is executed, control passes back to the top of the loop and the condition is tested again. This is repeated until the condition is false. Control then passes directly to the next (non-indented) statement.

Now you may see why PROMAL does not need statement numbers. **The structure of a PROMAL program is given by its indentation.** By the way, it is very easy to generate indented lines with the EDITOR. The TAB key (or CTRL-I) moves the margin in by one level of indentation (two spaces), and CTRL-Q (Apple) or CTRL-U (Commodore) moves it back out.

In our case, the WHILE tests to see if I is less than the length of the current line of interest, LINE. LENSTR is a built-in LIBRARY function which returns the length of a string, much like the BASIC function LEN.

Another built-in library function is CMPSTR, which is used in the next line:

```
IF CMPSTR(STRING,"=",LINE+I,TRUE,LENSTR(STRING))
```

CMPSTR compares two strings. It has the following form:

```
CMPSTR(string1,operator,string2,fold,limit)
```

Here string1 and string2 are the addresses of the two strings to be compared, and **operator** is the kind of comparison desired, chosen from:

```
"<", "<=", "<>", "=", ">=", ">"
```

which are the same as for BASIC. "Fold" should be TRUE if lower case letters are to be considered the same as the equivalent upper case letters. "Limit" is the maximum number of characters to compare.

The string comparison is done in an IF statement, which is much like an IF-THEN statement in BASIC. If the condition after the IF is true, then all the indented statements after the IF statement are executed. If the condition is false, then the indented statements are skipped. In our case there is only one indented statement. The IF statement above is roughly equivalent to the BASIC statement:

```
IF STRING$ = MID$(LINE$,I,LEN(STRING$)) THEN...
```

except that there is no provision in BASIC for treating equivalent upper and lower case letters as equal during a string comparison.

If our comparison is true, then the desired string (for example, "CA 95050") has been found somewhere in the line, so

```
RETURN TRUE
```

is executed, which exits from the function and returns the value TRUE to the calling routine. In PROMAL, FALSE is defined as a byte with the value 0, and TRUE is 1.

If the desired string is not found then:

```
I=I+1
```

is executed, which advances to the next character of the line, and the WHILE loop is repeated. If the desired string is not found anywhere in the line, then control falls out of the WHILE loop into:

```
RETURN FALSE
```

which simply exits back to the caller with the value FALSE returned. Therefore our function will return TRUE if LINE contains the string passed to it and FALSE if it doesn't.

Again, if some of this discussion seems unclear, don't worry. We are covering a lot of ground very fast and superficially, to try and give you "the big picture". Keeping this in mind, let's move on to the main program.

The main program starts after the last subroutine (there was only one in our case), and after the BEGIN. The first four lines are:

```
IF NCARG <> 2  ;wrong # of arguments?
  PUT NL,"FIND error: 2 args. needed."
  PUT NL,"Usage: FIND string file"
  ABORT
```

The variable NCARG is a companion to CARG, and is predefined in the LIBRARY. It tells the number of command-line arguments passed from the EXECUTIVE to the program. In our case, the FIND program needs two arguments, so we

check to see if two were given when FIND was executed. If not, we print two error messages and then ABORT. ABORT is a built-in procedure which returns control to the EXECUTIVE.

Assuming NCARG was 2 as it should be, the indented lines above would be skipped, and this line would be executed:

```
INFILE=OPEN(CARG[2])
```

OPEN is another built-in function, which opens a file. It expects the name of the file as its argument. In our case, we want to open whatever file name is the second argument on the command line. OPEN returns 0 if the open was **not** successful, and the "file handle" otherwise. We store this handle in INFILE. From now on, anytime we want to access the file, we use this file handle. You may have multiple files open at once.

First we must make sure the file was opened successfully. If not (for instance, if the file was not found), we just print an error message and quit:

```
IF INFILE=0    ;open error?
  PUT NL,"FIND error: can't open ",CARG[2]
  ABORT
```

Assuming the OPEN succeeded, we are now ready to search the file for lines containing our string:

```
WHILE GETLF(INFILE,LINE)    ; read a line
```

GETLF is another built-in function. It stands for "GET Line from File". The first argument is the file handle and the second argument is the address where we want the line in memory. In our case, we will read the line into the array LINE. GETLF returns TRUE if it was successful and FALSE if end-of-file was encountered before any data could be read. Since we are testing this returned value in a WHILE loop, the indented statements after the WHILE will be repeated until end-of-file is reached. These indented lines are:

```
IF HASSTRING(CARG[1]) ;string match?
  PUTF STDOUT,LINE,NL   ;yes, show
```

The IF statement calls our function, HASSTRING, passing it our desired string, which is the first command argument. IF HASSTRING returns TRUE, then the line contains the desired string and we should print it.

PUTF is another built-in procedure which is very similar to PUT. The only difference is that PUTF can output to a file, not just to the screen, like PUT. The first argument of PUTF is the file handle to write to. The remaining arguments are the same as for PUT.

Why do we want to use PUTF instead of PUT, and what is STDOUT? Well, we could have just used:

```
PUT LINE,NL  ;yes, show
```

instead, but then we would only be able to output to the screen. You might like to be able to output to the printer or a file, without having to change the program. This re-directing of output can be done using a PROMAL feature called **I/O Redirection.**

The LIBRARY pre-defines a file handle called **STDOUT** (STandarD OUTput). This file handle is always open. By default, it is opened to the screen. However, you can redirect it from the EXECUTIVE to the printer or to a file. For example, consider this EXECUTIVE command:

```
--> FIND "CA 95050" MAILLIST.T >P
```

This would redirect all the output to the printer ("P" is the name of the printer in PROMAL). This I/O redirection is done automatically by the EXECUTIVE. All your program has to do is output to STDOUT, and the output will go wherever the command line redirects it. It is not necessary to open this file, because the EXECUTIVE has already done it. The ">" is the output redirection operator of the EXECUTIVE and should follow the last argument. For example:

```
--> FIND "CA 95050" MAILLIST.T > CALLIST.T
```

would output the list of lines to a file called CALLIST.T

This completes our discussion of the FIND sample program. If you want to try the FIND program, you can exit the editor (f8 key followed by Q and <RETURN>), and then execute it from the EXECUTIVE. You don't have to compile it because the object file FIND.C is already provided (but you can if you want to). Since you don't actually have a mailing list on disk to try it on, you might want to try this example instead:

```
FIND "WHILE" FIND.S
```

What does this do?

If you'd like to try out the I/O redirection feature, but don't have a printer, try this:

```
FIND "NL" FIND.S >W
TYPE W
```

This will write all lines containing NL in the file FIND.S to the Workspace (deleting whatever was there before). The TYPE command will display the contents of the workspace on the screen. If you find the I/O redirection interesting, you might want to try this too:

```
DUMP 1100 1180 >W
TYPE W
```

What does this tell you about the built-in EXECUTIVE commands?

If you have persisted this far, you will have little difficulty learning to program your own applications in PROMAL. So far, you have learned how to EDIT and COMPILE a PROMAL program for handling text files.

REAL (FLOATING POINT) NUMBERS

So far the sample programs have dealt mostly with integer numbers, which is all that is needed for many applications. PROMAL also provides the data type REAL for floating point arithmetic. This is the kind of numeric data you know from BASIC, except that:

1. PROMAL real arithmetic is accurate to 11 significant digits instead of only 9 like BASIC.

2. You can precisely specify the output format for PROMAL real data (for example, how many decimal places you wish to use). This is very important for business applications, where decimal point alignment is expected:

BASIC OUTPUT	PROMAL OUTPUT
$100	$100.00
23.21	23.21
6.66666667	6.67
11.1	11.10
------	------
140.9766667	140.98

3. PROMAL real arithmetic is usually faster than BASIC (but not nearly so fast as arithmetic on BYTE, WORD, or INT data types).

4. The PROMAL LIBRARY does not include built-in functions for square root, trig functions, exponentials, log functions, etc. Instead, these functions are provided in source form which you can include easily in your programs as needed. **APPENDIX K** describes these functions.

A SIMPLE BUSINESS PROGRAM

A simple program called BUDGET, illustrating the use of REAL data is included on the PROMAL Diskette. To run the BUDGET program, type:

```
BUDGET
```

from the EXECUTIVE. This program displays a hypothetical budget report showing expenditures for a month. You can study the source file, BUDGET.S, with the EDITOR to see how to format REAL numeric output. We won't cover this program here, but the file BUDGETDOC.T is a text file which describes the program. The PROMAL LANGUAGE MANUAL provides full information on REAL data.

AN ADVANCED PROGRAM

Have you ever wondered how BASIC, PROMAL, or any language evaluates an arbitrary arithmetic expression with variables, constants, parentheses and operators? If so, you may have imagined that it takes a very complex program to do so. Well, the file CALC.S on your Demo disk contains a PROMAL program of about 180 lines (excluding comments) which simulates a four-function calculator with 26 "memories". This program can evaluate any arithmetic expression of arbitrary complexity, using the four operators +, -, *, and / plus parentheses. To try the program, type:

CALC

from the executive and just follow the directions. You may type in an expression, for example:

```
3.26+(1-.82)/3
```

CALC will display the answer (showing two digits after the decimal point by default):

```
 =          3.32
```

You may also type an assignment statement to one of the 26 variables, A through Z. For example:

```
p = 3.14
x = 2*p
```

You may also change the number of decimal places displayed for answers. To change to 6 decimal places, type:

```
#6
```

When you have finished experimenting with CALC, you can exit back to the EXECUTIVE by just pressing <RETURN> by itself. Now let's take a look at this program. Having a good-sized program will give us a chance to try out some of the more advanced editing features of PROMAL, too.

ADVANCED EDITOR FEATURES

From the EXECUTIVE, type:

UNLOAD
EDIT CALC.S

This will unload the programs we now have in memory and EDIT the source program for our four-function calculator. First, let's zip all the way down to the main program. Press the **F5** function key. The word FIND will appear in the status area below the working display area. Complete the FIND command like this:

FIND **'MAIN'**

and press RETURN. Be sure to remember the quotes (either " or ' will work, as long as they're the same on both ends of the string). The screen will almost instantly change to show the 20 lines beginning with:

```
BEGIN ; Main Program
```

The status line will show that this is line 244. The cursor will be on the **M** in **Main.** This is how the FIND command works. You can also search for a particular line by specifying the line number instead of the quoted string. For example, FIND 1 will put you back at the top of the program, and FIND 9999 will put you at the end of the program (because there are less than 9999

lines). You can also back up or go forward from where you are by using a signed number. For example, FIND -100 will back up 100 lines from your present cursor position and then re-display.

Now go back to the beginning of the program by using a **FIND 1** command. Do a **CTRL-N** to see the next 20 lines. The status line should indicate that the top of the screen is now showing line 21. The lines starting with CON declare some constants in the program. For example, the line:

```
CON LINESZ = 80 ; Max line size
```

defines the constant LINESZ to have the value 80 throughout the program. A constant is similar to a variable, but cannot have its value changed during execution. Below the constants are declarations for several variables. Press **CTRL-N** to advance to the next screen (starting with line 41). Notice the line:

```
REAL VAR[27]  ; Variables A-Z cur value
```

This array of type REAL will hold the current value for each of our simulated calculator's "memories". Suppose you decide you want to change the variable VAR to be called MEMS instead, throughout the program. Use the **F6** function key, and complete the command as follows:

CHANGE **100 'VAR' 'MEMS'**

and press RETURN. This tells the EDITOR to change 100 occurrences of VAR to MEMS. You will see the first occurrence of VAR highlighted (in a comment) and the status area will show the prompt:

CHANGE THIS STRING (Y/N/C=CANCEL)?

You can press Y to change the string and advance to the next occurrence, N to advance to the next occurrence without changing this one, or C to cancel the command at this point. Press **Y** and watch what happens. The next occurrence is highlighted, and you are again asked if you want to change it. However, this occurrence of VAR occurs in the word VARIABLES in a comment, so you don't want to change it. This is why you get a chance to "veto" each occurrence! Otherwise you might get some surprises. Press **N** to skip this occurrence. Continue pressing Y or N at each prompt, as appropriate (most will be N). When all occurrences have been found and presented to you, the status area will show:

11 CHANGES MADE. PRESS RETURN.

to indicate the total number of changes made out of the 100 you specified. Press <RETURN>. Now press **F5** to issue a **FIND 72** command. You should see a DATA statement that looks similar to:

```
DATA WORD HELP [] = ; Instructions...
"Please enter an arithmetic expression",
"such as \OF 3.14 * (20.25-8.5) \OE OR",
"an assignment statement to A through Z",
"such as \OF X=3-a/.55 \OE OR",
"#n to select n decimal places (0-8)",
"in answer (for example \OF #4 \OE) OR",
"just press RETURN to exit the program.",
" ", ; blank line
0  ; end of list
```

This DATA statement declares an array of strings called HELP. PROMAL DATA statements are somewhat like BASIC DATA statements, except that you don't have to READ the data into a variable; its already there. DATA statements are the only statements that may take up multiple lines. The brackets after the variable name indicate an array. It is not necessary to put a dimension inside the brackets for a DATA declaration, because PROMAL will figure out how big to make the array. The first subscript of a PROMAL array is always 0, not 1. Therefore if we later have a statement:

```
PUT HELP[2]
```

it will display:

```
an assignment statement to A through Z
```

If you have a Commodore 64 instead of an Apple, the "\0F" and "\0E" in the DATA statements will be replaced by **£**12 and **£**92. The Commodore does not have a backslash key, so the "pounds sterling" key is used instead. The \ or **£** symbol is used in a string to embed non-printable characters. The \ is followed by exactly two hexadecimal digits which give the code for the desired embedded control character. In this case, the \0F and \0E turn reverse video on and off respectively on the Apple (and **£**12 and **£**92 do the same function on the Commodore 64).

Note that the array HELP is declared to have a type of **WORD** rather than **BYTE** as you might have expected. This is because the array is actually an array of **pointers** to the strings, and each pointer is a word (this is explained in the PROMAL LANGUAGE MANUAL).

Getting back to the EDIT session, suppose you decide that you want to move the lines containing the DATA statement for the HELP array closer to the top of the program for some reason. Put the cursor on the first line of the DATA statement (line 73) and press the MARK function key (**F3**). The line will be highlighted. Now move the cursor to the last line of the DATA statement (line 82) and press **F3** again. The lines will be shown in reverse video, indicating they are now "marked" for some action. The function key legends in the status now show the choices for what to do with the marked lines:

```
1=DELETE 2=       3=MARK   4=WRITE  5=FIND   6=MOVE   7=COPY   8=CANCEL
```

COPY makes a copy of the marked lines at the new cursor location. WRITE lets you write out the marked lines to a file, the printer, or the Workspace. MOVE cuts the lines from where they are and inserts them at the cursor location.

DELETE simply deletes the marked lines. CANCEL lets you back out of the command without doing anything. Move the cursor up, scrolling the screen, until the cursor is on line 32, right below INCLUDE LIBRARY. Then press the MOVE function key (**F6**). Instantly you will see the screen re-displayed with the DATA statement moved to the new location, and the function key legends restored to normal.

You have now used all the function keys except F1 (DELETE LINE), F2 (INSERT LINE), and F4 (RECALL). You can experiment with F1 and F2 to see how they work. The RECALL function key (F4) is used to insert another file into your program at the cursor location. You can "cut" from one program and "paste" into another using WRITE and RECALL. These commands are described in the EDITOR section of the PROMAL USERS GUIDE.

Let's try one more thing with the EDITOR. **FIND** line **258.** The second line will begin a WHILE loop. Now suppose that for some reason you decide you need to add another loop of some kind that would encompass all the lines in the existing WHILE loop. This means that you need to indent all these lines by another level. Here's an easy way to do this. Put the cursor on the first line to be indented (the WHILE statement) and press **CTRL-^** (control key with shift and 6) if you have an Apple or **CTRL-J** if you have a Commodore 64. The line jumps to the right by two spaces. Press the same key again and the next line will indent. Just repeat this for as many lines as desired. To get rid of an unwanted level of indentation, press CTRL-O (the letter O). If you forget what control keys do what, use the HELP function key (F7).

You've now seen most of the major features of the EDITOR. If you wish you may study the rest of the program before exiting back to the EXECUTIVE. The comments should give you an idea what is going on. An important routine is procedure GETTOKEN. This routine reads characters from the line until it has a complete "token". A token is a complete number, a variable name, or an operator. When examining this routine, it will be helpful to know that the PROMAL operator @< is called an indirect operator. It follows a variable name that is being used as a pointer. LPTR@< therefore gets the character at the address given by LPTR. PROMAL supports other indirect operators as well, and can perform very powerful operations using pointers.

This routine uses several new functions defined in the LIBRARY. The standard function TOUPPER converts lower case characters to upper case. The standard function INSET determines if a character is in a string or set of characters. The function NUMERIC tests if a character is a numeric digit. The function STRREAL converts a string to a real value (somewhat like the BASIC function VAL). All these functions are detailed in the PROMAL LIBRARY MANUAL.

The actual parsing of the input to CALC is done by a technique known as "recursive descent". This is a goal-oriented technique which tries to match the input line to a known pattern, where each part of the pattern is processed by a subroutine. The theory involved is fairly advanced, and there is no need for you to understand it. If you are interested you can follow through how the CALC program processes a sample expression "by hand", which will be very instructive to understanding it. Also, Appendix P of the PROMAL LANGUAGE MANUAL has some further discussion of recursive descent parsing and syntax diagrams. This will only be of interest to the very advanced programmer, however. Our main purpose in examining this program is to show how to use the advanced editor features, and introduce some new statements. Also, we wanted

to give credence to our claim that PROMAL is a suitable tool for developing compilers, assemblers, and other system programs.

Exit the EDITOR with function key **F8.** Don't save the modified program to workspace or to disk. Instead, simply exit back to the EXECUTIVE by pressing **Q** and <RETURN>. If you wish you may compile the original CALC program directly from disk, by typing the EXECUTIVE command:

```
COMPILE CALC
```

The COMPILER will ask you if you want to replace the existing CALC.C file when it finishes. You can reply either Y or N, since the result will be the same because you haven't changed the program.

SOME SPECIAL CAPABILITIES

For advanced programmers, here are some additional capabilites which may be important to you. These are described in the PROMAL LANGUAGE MANUAL:

1. You can assign the address of an **external** variable anywhere in memory. This allows you to give meaningful names to those "special" addresses. For example on an Apple system you might use:

```
EXT BYTE HIRES_ON AT $C057
```

which assigns the name HIRES_ON to the Apple Softswitch controlling hi-res graphics. The statement HIRES_ON=1 will therefore enable hi-res mode. A Commodore 64 example would be:

```
CON YELLOW=7
EXT BYTE BACKGROUND AT $D021
...
BACKGROUND = YELLOW
```

This sequence lets you give a meaningful name to the VIC background color register and manipulate it like any other variable. Isn't BACKGROUND=YELLOW much clearer than its BASIC equivalent of POKE 53281,7?

2. You can perform bit-level operations with PROMAL such as AND, OR, EXCLUSIVE OR, and SHIFTS. This often eliminates the need for machine language programming for I-O or special needs.

3. If you ever do need machine language routines, PROMAL provides a clean interface. You can call machine language routines from PROMAL with passed arguments (you can even set the hardware registers if you want). You can embed machine language routines in DATA statements, or load larger programs from disk using a built-in library function, or directly from the EXECUTIVE.

SOME SPECIAL SYSTEM-DEPENDENT DEMO PROGRAMS

Your PROMAL Demo disk has some other demonstration programs which exploit the special capabilities of your computer. You may wish to try these programs, which are described briefly below.

FOR THE APPLE II ONLY:

The following section applies only to the Apple II version of the PROMAL Demo diskette. If you have a Commodore computer, you may wish to skip down to the section, "FOR THE COMMODORE 64 ONLY".

A DATABASE APPLICATION PROGRAM

Now that you've seen some of the features of the PROMAL system in action, let's move on to a more complex application program. This will give you the chance to see more capabilities of the PROMAL language and to use some advanced editing features. This example takes advantage of the 80 column screen on the Apple IIe or IIc.

A hypothetical record store (Pete Promal's Record Shop) keeps a database of information about what albums are in stock on disk. The records are kept as ordinary sequential text files, with each record having the following format:

Field offset (size)

0 (19)	20 (10)	30 (20)	50 (10)	60 (2)	63 (4)	68 (4)
Artist name	(First)	Album name	Label	Year	Quantity	Bin #

For example, a typical record from the database file looks like this:

```
Jackson            Michael   Thriller            Epic      82 0040 0108
```

The last two columns tell the quantity on hand and the bin number, which is the physical location of the albums in the store.

A small segment of this hypothetical database is the file RECORDDATA.T on the demo disk. To see what it looks like, type:

TYPE RECORDDATA.T

The SORTDEMO program lets you sort this database by any of the fields, in ascending or descending order. Type:

SORTDEMO

from the EXECUTIVE and you should see:

```
                    PETE PROMAL'S RECORD SHOP

                           SORT UTILITY

Please select the sort options below.

<space>   changes the highlighted option.
<RETURN>  accepts the highlighted option.
<ESC>     exits the program.

Sort order is [ASCENDING]
```

If you press <space>, the highlighted word [ASCENDING] turns to [DESCENDING]. Pressing <space> again changes it back. Press <RETURN> to accept an ascending sort. In a similar manner, you can pick the remaining options to choose which field of the record to sort on and what output device to select (screen, disk, or printer).

After selecting the output device, the program will read the data file, sort the records in the manner you specified, and display the records in sorted order. The program then exits to the EXECUTIVE.

We will not go over this source file for this program, but if you wish to study it, the comments will help explain the operation of the program.

FOR THE COMMODORE 64 ONLY:

The following section applies only to the Commodore 64. If you don't have a Commodore 64, you may wish to skip down to the section, "IN CONCLUSION".

SPRITES, ANIMATION, AND SOUND SYNTHESIS WITH PROMAL

On the Commodore 64 PROMAL Diskette is a moderately complex program called BILLIARDS. This illustrates how PROMAL can be used to program sprites and sound in real time. Although the program is not a complete game program, it could be upgraded to be one. The program was deliberately chosen because it involves a lot of computation, a great deal more than most animated games. The program simulates the motion of three balls on a billiard table. It actually approximates the true behavior of the balls by the physical equations of motion. It solves collisions between balls and the rails using transfer of momentum, and includes coefficients of drag so the balls appear to slow down realistically. We're not saying you have to be a physicist to use PROMAL; but rather, that if you **do** need to do something complex, PROMAL can handle it a lot better than BASIC.

If you write this program in BASIC, there simply won't be any animation to speak of, because the balls will move slowly and jerkily. With PROMAL, the balls bound around the table fairly realistically, complete with sound effects. The billiards program has already been compiled for you on the PROMAL Diskette. To try it from the EXECUTIVE, type:

UNLOAD
BILLIARDS

After the program loads, you will see the table and three balls. One of the balls will zip diagonally across the table, colliding with the others. When the balls almost stop, the program exits back to the EXECUTIVE. You can use CTRL-B if you want to run it again. After you've learned more about PROMAL, you might like to modify this program to accept different angles and velocities for the "hit" on the cue ball. Or, if the balls on the screen look egg-shaped instead of round (due to the fact that pixels are not the same width as their height), you may want to change the sprite data to make the balls look more round.

We won't go over the source program for BILLIARDS, but you may examine it with the EDITOR if you wish. Although the comments will give you a good idea what is going on, you will need to study the PROMAL LANGUAGE MANUAL to fully understand the details of this program.

If you liked the BILLIARDS program, try this:

UNLOAD
INFILTRATOR

This will execute a fairly simple arcade-type game written entirely in PROMAL. The source code to this program is included on one of the PROMAL disks, if you'd like to examine or improve it. This game features smooth horizontal scrolling (impossible from BASIC), multiple multi-color sprites, and some "phasor" and explosion sound effects. Have fun!

Note: Because this program's source file is larger than 400 lines, you will not be able to compile it with the Demo compiler. The standard compiler can compile it easily with the B option. This program uses an INCLUDE file.

IN CONCLUSION

As we said earlier, the purpose of this manual was not to teach you how to program in PROMAL but to give you a good idea about what it is like to program in PROMAL. You have now gone through the mechanics of editing, compiling and running small and large programs. You have used a few EXECUTIVE commands and know how the EXECUTIVE can run programs and pass command arguments to programs. We hope you now have a good idea of what PROMAL is all about.

There are many powerful PROMAL features we have not even touched on yet. For example, you can use the EDITOR to prepare a "script" of commands for the EXECUTIVE to execute using the JOB command. This can be used to run a whole series of programs or commands as a "batch". Also, we have only seen a very few of the routines in the LIBRARY. There are many routines in it to do everything from a block-move to searching a linked list.

The PROMAL LANGUAGE MANUAL contains a full explanation of how to program in PROMAL, with plenty of examples provided. The LIBRARY has a reference manual of its own. The PROMAL USER'S GUIDE explains all the built-in EXECUTIVE commands in detail, as well as all EDITOR commands and compiler options. If you've liked what you've seen of PROMAL, but don't quite understand everything we've covered, you'll still have very little difficulty becoming a proficient PROMAL programmer.

MAKING WORKING DISKS

To start using PROMAL for writing your own programs, you will want to make a "working disk" with just the files you need on it plus your own programs. **Appendix O** tells you what files you need and how to copy them.

END USER VERSION AND DEVELOPER'S VERSION

The PROMAL system is available in two versions. The Standard or "End User" version gives you everything you need to develop programs for use on your computer, or for use on other people's computers which also have PROMAL. The Developer's Version includes all of the Standard System, but in addition contains a special Utility which will let you generate stand-alone PROMAL applications which will run on computers that do not have PROMAL. The developer's Version includes an Unlimited License for you to distribute these programs without payment or royalties of any kind to SMA. It also includes another small manual, the PROMAL DEVELOPER'S GUIDE. If you have purchased the End User system, you can upgrade later to the Developer's Version by just paying the difference in price (call for details).

SOURCE CODE FOR THE PROMAL SYSTEM

As an option, you can purchase the source code for the PROMAL EXECUTIVE, EDITor, and a source listing of the assembly language Runtime Package and Library. This is a unique benefit to PROMAL programmers, which is not available for any other commercial programming system. Contact SMA for pricing and availability.

GRAPHICS TOOLBOX

A Graphics Toolbox is available as an option. This provides very fast, high resolution drawing subroutines you can use from your PROMAL program. It can be used with either the End User or Developer's version of PROMAL.

With this package you can easily write programs to draw bar graphics, pie charts, function plots, and other graphic images. The Graphics Toolbox for the Commodore 64 draws in 16 colors using the 320 by 200 high resolution mode. On the Apple, graphics are done in 280 by 192 high resolution monochrome, since the Apple does not have a high-res color mode. Despite substantial underlying differences in hardware, graphics applications written for the Commodore are highly portable to the Apple and visa versa.

Call SMA for ordering information on the low-cost Graphics Toolbox.

REGISTERING YOUR PROMAL SYSTEM

After you've opened your sealed System diskette(s), be sure and fill out and send in your "END USER AGREEMENT, LICENSE and REGISTRATION FORM". This is the **only** way you will be able to get on our mailing list, so you can receive important upgrade notices, product announcements and the PROMAL NEWSLETTER.

CUSTOMER SERVICE

If you run into a problem with PROMAL you can't resolve by carefully reading the manual (and the trouble shooting guide in Appendix C), please call our Customer Service line at **(919) 878-3600.** Please be prepared to tell us what your computer hardware is, your version of PROMAL, serial number (on sealed disk if you've opened it), and an **exact** description of what actions you took and what symptoms you observe. Please get the exact wording of any error messages. This will make it much easier for us to help you.

Thank you for purchasing PROMAL. We are sure you will find it to be an indispensible addition to your software collection.

P R O M A L

(PROgrammer's Micro Application Language)

USER'S GUIDE

A GUIDE TO USING THE PROMAL

-- EXECUTIVE --
-- EDITOR --
-- COMPILER --

For Apple II and Commodore 64 Computers

SYSTEMS MANAGEMENT ASSOCIATES, INC.
3325 Executive Drive
Raleigh, North Carolina 27609
(919) 878-3600

Rev. C - September 1986

PROMAL USER'S GUIDE

INTRODUCTION

Welcome to the exciting world of programming in PROMAL. PROMAL provides a **complete programming environment** to let you get the most out of your computer. Unlike other programming languages which have historically come from big computers and have been "shoehorned" into personal computers, PROMAL was designed from the ground up for use on small machines. Because of this, it provides the tools you need to quickly and easily write programs which run "lightning-fast".

Not only do PROMAL programs often run 20 to 100 times (or more) faster than BASIC, but non-trivial programs are usually easier to program and maintain in PROMAL than BASIC.

Your PROMAL programming system includes all of the following:

* An operating system EXECUTIVE for interactive control
* A powerful full-screen program EDITOR for preparing programs
* A fast, one-pass COMPILER for the PROMAL language
* A standard LIBRARY of over 50 versatile subroutines, ready to use
* Ready-to-run demonstration programs for you to run, study, and modify
* A complete manual with examples to guide you

Let's take a quick look at what each of these tools does for you.

The PROMAL operating system **EXECUTIVE** is your control center. You type in commands to run programs, activate the PROMAL editor or compiler, and perform other operations. A number of built-in commands are provided for manipulating files, displaying and changing memory, etc. In addition, you can add your own commands. The PROMAL EXECUTIVE lets you have several programs in memory at once, and you can run any of them instantly by merely typing the name of the program.

The PROMAL full screen **EDITOR** makes it easy to create or change programs or text information. The editor is like a good word processor, except that it is designed specifically for writing PROMAL programs. To make changes, you simply move the cursor around on the screen and insert or delete text as desired. The editor can scroll forwards or backwards to rapidly display any part of your program. Powerful search and replace commands make it easy to make corrections. You can "cut and paste" blocks of text, too.

The most important part of the system is the PROMAL language and compiler. The **COMPILER** takes the program you created with the EDITOR and converts it to a form which runs with nearly the speed of assembled machine language programs. It is this compilation process which is mainly responsible for PROMAL's great speed. Compiled PROMAL programs also occupy less memory than BASIC or other languages. You can save your compiled program on disk, and it can be run at any time later by just typing its name.

The **LIBRARY** of over 50 subroutines is already built into the PROMAL system. You can call these routines from your program to perform a wide variety of tasks. Having all these subroutines immediately available greatly simplifies the job of programming.

There's nothing like some good examples to speed the learning process, so several complete, useful **demonstration programs** are included on the PROMAL disk.

ABOUT YOUR PROMAL MANUALS

If you have not already done so, you should read your **MEET PROMAL!** manual, which will provide you with a "hands-on" guided tour of the PROMAL system as a whole. This short manual introduces many of the novel concepts of the PROMAL system in a "get-acquainted" style.

This manual, the PROMAL USER'S GUIDE, tells you how to use the various components of the PROMAL system. After this introduction, it is divided into three major sections, covering:

1). Operation of the EXECUTIVE (including all EXECUTIVE commands);
2). Operation of the EDITOR;
3). Operation of the COMPILER.

The PROMAL LANGUAGE MANUAL, describes the PROMAL language in detail. You will probably want to read it after you have gained some proficiency with the EDITOR and EXECUTIVE by reading the USER'S GUIDE.

The PROMAL LIBRARY MANUAL provides a detailed reference for the the subroutines in the LIBRARY, arranged alphabetically. You will want to skim this material after reading the LANGUAGE MANUAL, and refer back to it for details as the need arises.

A set of Appendices serves all of these manuals, giving supplemementary information. An index is provided to all manuals and the Appenidces.

If you purchased the Developer's Package, you will have an additional manual, the DEVELOPER'S GUIDE, which covers topics relevent to making stand-alone PROMAL programs which can be run on systems without PROMAL.

STARTING THE SYSTEM

The **MEET PROMAL!** manual tells you how to "boot up your system" using a copy of the Demo disk. You should always use a "working disk" that has a copy of the PROMAL software on it, and leave the original disk in a safe place.

Once you have your system booted up, PROMAL will sign on and the EXECUTIVE will now display the default meanings of the function keys (note: see the JOB command description below for a way to defeat the display of the default function keys during boot-up). This display will look similar to this:

Commodore	Apple II
F1 = EDIT	F1 = EDIT
F2 = DUMP	F2 = PREFIX *
F3 = COMPILE	F3 = COMPILE
F4 = GET	F4 = GET
F5 = FILES	F5 = FILES
F6 = MAP	F6 = EXTDIR *
F7 = HELP	F7 = HELP
F8 = COPY	F8 = COPY

Note: F4 means hold either Apple key and press 4.

If you have read **MEET PROMAL!**, you already know that pressing one of the function keys is equivalent to typing in the command name it stands for. For example, pressing F1 will cause the word "EDIT" to appear on the screen, just as if you typed it in. The "-->" is the PROMAL EXECUTIVE prompt, followed by a blinking cursor. This indicates that the EXECUTIVE is waiting for you to type in a command.

You are now ready to begin using PROMAL.

EXECUTIVE COMMANDS AND COMMAND EDITING

To tell the EXECUTIVE something, you can use any of the function keys or type in a command that the EXECUTIVE knows. The "built-in" commands are described in detail in the next section. All commands must be terminated by the RETURN key. Up until the time you press the RETURN key, you can use any of the line-editing keys described in **Table 1** to make corrections.

Once you press RETURN, the EXECUTIVE will attempt to execute whatever command you have typed. There are three "levels" of commands, which the EXECUTIVE searches in this order:

1). Built-in commands
2). User-defined commands in memory
3). User-defined commands on disk

If the EXECUTIVE can not find the command in any of these places, it will display:

```
*** ERROR: PROGRAM OR OVERLAY NOT FOUND: xxxx
--> _
```

User-defined commands are simply compiled PROMAL programs. Several of these command files are on the Demo disk. They are easily recognized because their names end in ".C", indicating a command file. You can create your own commands by writing a PROMAL program and compiling it. To run a PROMAL program, you don't have to LOAD it and RUN it like you do with BASIC; you simply type its name. The EDITOR and COMPILER sections of this manual and the LANGUAGE MANUAL contain all the information you need to create your own commands.

Commands may be typed in either upper or lower case letters. PROMAL operates using upper-and-lower case, like a normal typewriter. If you prefer all upper case letters, you can press CTRL A, which enables alpha-lock.

TABLE 1

PROMAL LINE-EDITING KEYS

Commodore Key	Apple Key	Description
RETURN	**RETURN**	End of line. The completed line is entered into the PROMAL system. May be typed from any cursor position in the line. Maximum line size is 80 characters for the EXECUTIVE.
DEL	**DELETE**	Replace the character left of the cursor with a blank and back up the cursor one position
INST	**CTRL E**	Enable "insert mode". Any characters subsequently typed will be inserted before the character the cursor is on, pushing any existing text to the right. Exit insert mode by pressing RETURN or other line-editing keys.
CTRL <--	**CTRL D**	Delete character with pullback. Deletes the character under the cursor and pulls any remaining text to the left to fill in the gap.
==>	**-->**	Cursor right. Moves the cursor to the right, without altering the character under the cursor. Stops at the end of the line. Repeats automatically after a brief pause if held down.
<==	**<--**	Cursor left. Moves the cursor to the left, without altering the character under the cursor. Stops at the first character entered. Repeats automatically after a brief pause if held down.
CTRL X	**CTRL X**	Cancel the entire line. Erases all characters typed on the line and repositions the cursor to the first character position.
CTRL K	**CTRL **	Clear to end of line. Erases all characters from the cursor to the end of line.
CTRL Y	**CTRL L**	Jump to last character of line. Moves the cursor to the column after the last character on the line, without affecting the line content.

NOTE: The CTRL key is used like a shift key. CTRL X means you hold down the CTRL key and press X at the same time. N/A means not available.

TABLE 1 (continued)

Commodore Key	Apple Key	Description
CTRL [	**CTRL F**	Jump to first character of line. Moves the cursor to the first character position, without affecting the line content.
CTRL A	**CTRL A**	Toggle Alpha-Lock mode. When pressed the first time, causes all subsequent alphabetic characters to be entered and displayed as upper case when typed. Does not affect other characters displayed on the screen or already typed. Pressing CTRL A again returns to normal upper and lower case alpha mode.
F1 through F8	**Apple 1 through Apple 8**	Function key. Erases current text on command line and enters the equivalent function key definition. More can be typed after the function key definition or it can be edited further if desired.
CTRL B	**CTRL B**	Backtrack. Erases current text on the command line and enters the last line entered. More can be typed after the recalled command, or it can be edited further. Pressing CTRL B again will recall the next-to-last command entered. This can be repeated up to the limit of the backtrack buffer of 256 characters; then the display will "wrap around" to repeat the most recent command again.
CTRL STOP	**CTRL RESET**	Program abort. Unconditionally aborts the currently-running program and returns control to the PROMAL EXECUTIVE, closing any open files. Should be used only as an "emergency exit" (does not work if interrupts are disabled or if a machine-language program has executed an illegal opcode). Apple version clears screen and may cause loss of data in files open for writing.
CTRL Z	**CTRL Z**	Indicates end of file from the keyboard device if it is the first character of a line. See the TYPE command for an application.

NOTE: The CTRL key is used like a shift key. CTRL X means you hold down the CTRL key and press X at the same time. N/A means not available.

It is possible to alter almost all of the choices for editing keys (See **Appendix G**). The default keys were chosen so as not to conflict with pre-defined Commodore keys.

For the Commodore 64, CTRL-STOP will not be operational while the disk is being accessed with DYNODISK enabled, because DYNODISK disables interrupts temporarily while it is running.

TABLE 1 (continued)

SPECIAL SYSTEM-DEPENDENT KEYS

Commodore Key	Apple Key	Description
STOP	**N/A**	Stop action. This key temporarily suspends all program execution (including user-defined programs, built in commands, and machine-language programs) until the key is released. It is useful for halting a rapidly-changing display so you can read it (does not work if interrupts are disabled).
CTRL	**N/A**	Slow display. During display on the screen, slows down the scroll rate while held down. Makes it easier to read rapidly scrolling text.
N/A	**CTRL C**	Abort command. Can be used to abort an Executive command or a running program if it is displaying on the screen. This method is preferred to CTRL-RESET for aborting programs on the Apple.
N/A	**CTRL S**	Pause output. Temporarily halts display to screen until any key is pressed.

NOTE: The CTRL key is used like a shift key. CTRL X means you hold down the CTRL key and press X at the same time. N/A means not available.

BUILT-IN EXECUTIVE COMMANDS

The EXECUTIVE has a number of built-in commands which are always available, summarized in **Table 2** below. These commands are explained in detail in the following sections.

TABLE 2

BUILT-IN EXECUTIVE COMMANDS

Command	Avail*	Function
BUFFERS	**A**	Set the number of ProDOS disk buffers (# open files).
COLOR	**C**	Change the current color and/or screen background color.
COPY	**AC**	Copy a file.
CS	**AC**	Clear the screen.
DATE	**AC**	Change the current date (See note).
DELETE	**AC**	Delete a file.
DISKCMD	**C**	Send C-64 disk commands and display error channel replies.
DUMP	**AC**	Display memory in hexadecimal and ASCII characters.
DYNO	**C**	Enable/disable double speed read for 1541/1571 disk drives.
EDIT	**AC**	Enter the full screen PROMAL EDITOR.
FILES	**AC**	Display the names of files on disk.
FILL	**AC**	Fill a region of memory with a constant.
FKEY	**AC**	Redefine a function key or display present assignments.
GET	**AC**	Load a PROMAL or machine language program into memory.
GO	**AC**	Execute a machine language program in memory.
HELP	**AC**	Display a "help" menu of EXECUTIVE commands and control keys.
JOB	**AC**	Execute a list of EXECUTIVE commands stored in a file.
LOCK	**A**	Lock (write-protect) a file.
MACRO	**AC**	Define a command macro.
MAP	**AC**	Display the current memory allocation and loaded programs.
NEWDIR	**A**	Create a new directory.
NOREAL	**AC**	Discard support for REAL data (makes more memory available).
PAUSE	**AC**	Display a message and wait for RETURN key.
PREFIX	**A**	Display or change disk volume name or subdirectory.
QUIT	**AC**	Exit to BASIC (Commodore) or to specified system (Apple).
RENAME	**AC**	Change the name of a file.
SET	**AC**	Set memory locations to specified values or characters.
SIZE	**AC**	Display the size of a compiled PROMAL program.
TYPE	**AC**	Display a file of text on the screen, printer, etc.
UNLOAD	**AC**	Remove a PROMAL program from memory.
UNLOCK	**A**	Unlock (allow writing) a file.
WS	**AC**	Clear or alter the size of the Workspace (in-memory file).

***Note:** **A**=Apple, **C**=Commodore
DATE is not a built-in command on the Commodore 64, but a compiled program which is automatically unloaded after it is run.

ARGUMENTS FOR EXECUTIVE COMMANDS

Many EXECUTIVE commands have required or optional **arguments.** An argument is a series of characters separated from the command by one or more blanks. For example:

```
COPY MYFILE.T
```

has one argument, "MYFILE.T", and

```
DUMP 1000 1078
```

has two arguments. The kind of argument needed (if any) varies with the individual commands. Frequently an argument will be a file name to operate on.

In order to describe what kind of arguments a command can have, the following notation is used in this manual:

(1). A name shown in all CAPITAL letters indicates the name of the command or a word which must be typed in exactly as shown. It may be typed in either upper or lower case letters.

(2). A name shown in Upper and lower case letters is a description of something the user must type in. For example,

```
COPY Filename
```

means the argument must be a legal PROMAL filename.

(3). Anything enclosed in square brackets, "[ ]" is optional. The text will explain what the default is if the optional argument is not specified.

(4). Ellipsis (...) are used to show an arbitrary number of repetitions of the preceding item. For example:

```
SET Address Value [...]
```

means that the SET command can have an arbitrary number of Value arguments specified.

FILE NAMES

File names are frequently used as arguments for EXECUTIVE commands. File name requirements differ somewhat for various computers, because the disk formats and underlying operating systems are different. In order to promote portability between computers, PROMAL uses a default naming convention that is very similar for all computers, but may not allow access to all file names and file types that are legal on a particular computer. PROMAL normally operates on these **PROMAL files** automatically, by default. However, provision is made in the EXECUTIVE and PROMAL language to be able to operate on **any** type of file which is legal on your computer. Find the file name rules which apply to your computer below. Hereafter, any reference to a file name means a PROMAL file name unless otherwise stated.

File Names for **Commodore 64** Computers

PROMAL can operate on standard **PROMAL files**, or any kind of file available on the Commodore 64. By default, PROMAL operates on standard PROMAL files, which conform to these rules:

(1). The name must be 1 to 14 characters long, with the first character being an alphabetic character. If a single character name is chosen, it should not duplicate the names of any of the PROMAL devices given in **Table 4.** The remaining characters must be alphabetic, numeric, or the left-pointing arrow character (an ASCII underline character; this is the key above the CTRL key). The name may NOT contain blanks or other punctuation, and is **not** enclosed in quotes when used as a EXECUTIVE command argument. Names may be typed in upper or lower case, but are converted to all upper case internally.

(2). The name part can optionally be followed by a period and a single character **file extension.** The file extension must alphabetic or numeric. It indicates the "kind" of file. **If omitted, a default extension of ".C" will be assumed, which indicates a PROMAL command file (executable program).** Multiple character file extensions may be used but are not recommended.

(3). The file name may have an optional drive number prefix followed by a colon. The choices are **0:** or **1:**. If no prefix is specified, 0: is assumed. See **Appendix N** for information on multiple drive systems.

All normal PROMAL files on the Commodore 64 are stored as sequential (SEQ) type files, including executable programs. PRG and REL files are not normally used. The following are examples of legal PROMAL file names:

```
AB         x7         MyData      YOUR45.T    DOIT.C
Hello_There.C         T12345.S    Z_          RECOVER.S
0:STUFF               1:AUXDATA.D
```

The names above which do not have a file extension will be stored on disk with the file extension ".C". Thus if you type AB for a file name, the file AB.C will be the file acted upon (EXCEPTION: The EDITOR and COMPILER will assume a file extension of .S for the source files they operate on). The following names are not legal PROMAL files for the reasons noted in parentheses:

```
7THDATA.S          (must start with an alphabetic character)
MY Data            (can't have embedded blanks or punctuation)
THE_LAST_PROGRAM   (can't have more than 14 characters)
OLD.STUFF.D        ("." can only be used to start the file extension)
```

EXECUTIVE commands normally use PROMAL file names. However, you may access any file name which is legal on the Commodore 64 by enclosing the name in quotes. The COPY command also requires specification of the file type for non-SEQ files, as is described later.

Appendix **M** describes PROMAL support of relative files, which should not be manipulated with EXECUTIVE commands. It is also possible to open files to access Commodore direct access files, directories, and the command/error channel. These facilities are described later.

File Names for **Apple II** Computers

PROMAL can operate on standard PROMAL files, or any kind of file available under ProDOS. By default, PROMAL operates on standard PROMAL files. A PROMAL file name is slightly more restrictive than a ProDOS file name. Legal PROMAL file names must conform to these rules:

(1). The name must be 1 to 13 characters long, with the first character being an alphabetic character. The remaining characters must be alphabetic, or numeric. The name may NOT contain blanks, periods, underlines, or other punctuation. If a single character name is chosen, it should not duplicate a PROMAL device name given in **Table 4.** Names may be upper or lower case. ProDOS converts all names to uppercase internally.

(2). The name part can optionally be followed by a period and a single character **file extension.** The file extension must alphabetic or numeric. The file extension indicates the "kind" of file. If omitted, **a default extension of ".C" will be assumed,** which indicates a PROMAL command file (executable program). Multiple character file extensions are permitted but not recommended. The total name including the extension may not exceed 15 characters.

(3). The file name may have an optional **pathname** which specifies the ProDOS volume name and/or subdirectory name. Volume names are indicated by a leading / character, are up to 15 characters long, and start with an alphabetic character. Volume names are assigned when the disk is formatted using the ProDOS Utility. Subdirectories follow the same naming rules as volumes, and can be specified when files are created. **If no prefix is specified as part of the file name, then the current prefix will be used.** The current prefix is set by the PROMAL **PREFIX** command, and is initially the prefix of the disk or directory from which PROMAL was booted. The total combined pathname and filename cannot exceed 60 characters. For floppy disks, we suggest you avoid using subdirectories. If a path name and file is specified without a leading / character, it will be appended to the present path. For example if the present prefix is /MYDISK/, and a name given is PROGS/GO.C, then the resulting path will be /MYDISK/PROGS/GO.C.

(4). In lieu of a volume name, you may use a two character drive prefix as follows:

```
0:   The /RAM volume (slot 3 drive 2)
1:   Floppy drive 1 (slot 6)
2:   Floppy drive 2 (slot 6)
```

When using the drive prefix, PROMAL will read the volume name from the selected drive and use it for the volume name part of the file name. It does not change the current prefix.

The following are examples of legal PROMAL file names on the Apple II:

```
AB          1:x7       MyData       YOUR45.T       DOIT.C
HelloThere.C           T12345.S     0:ZZ.SYSTEM    RECOVER.S
2:/WORK1/STUFF         /USER.DISK/MATH/DATA.D      X
```

The names above which do not have a file extension will be stored on disk with the file extension ".C". Thus if you type AB for a file name, the file AB.C will be the file acted upon (EXCEPTION: The EDITOR and COMPILER will assume a file extension of .S for the source files they operate on). The following names are ILLEGAL on the Apple II for the reasons noted in parentheses:

7THDATA.S	(must start with an alphabetic character)
0:MY_Data	(can't have embedded underline, blanks or punctuation)
OLD.STUFF.D	("." can only be used to start the file extension)

EXECUTIVE commands normally use PROMAL file names. However, you may access any file name which is legal with Apple II ProDOS by enclosing the name in quotes. Placing the name in quotes suppresses the default file extension. This allows you to use the EXECUTIVE to copy, delete, and rename files created by BASIC, word processors, or other non-PROMAL programs. For example:

```
COPY "PRODOS"
```

is an EXECUTIVE command to copy the PRODOS system file (not PRODOS.**C**).

FILE EXTENSIONS (ALL COMPUTERS)

Table 3 below lists the customary file extensions used for various kinds of PROMAL files. Other extensions may be devised by the user for special needs.

TABLE 3

PROMAL FILE EXTENSIONS

Extension	Type of file indicated
.C	A Command file. An executable (compiled) PROMAL program. This is the default extension.
.D	A data file.
.E	A PROMAL EXPORT file (for separate compilation, described in Chapter 8 of the LANGUAGE MANUAL).
.J	A "Job" file, usually prepared with the EDITOR, used to drive the EXECUTIVE from a script of commands.
.L	A program listing
.R	A Commodore 64 relative file (see Appendix **M**)
.S	A PROMAL source program, normally prepared by the EDITOR.
.T	A text file, other than a PROMAL source program.
.X	A cross-reference map (output from XREF utility).

NUMERIC ARGUMENTS

Some EXECUTIVE commands accept numbers for arguments. The built-in EXECUTIVE commands all require numeric arguments to be specified in **hexadecimal.** User-defined programs may specify decimal or hexadecimal at the discretion of the programmer. It is not necessary to understand hexadecimal numbers for casual use of the EXECUTIVE. The Commodore and Apple Reference Manuals describe hexadecimal notation. Hex numbers may be specified with any number of

digits; however, the value must not exceed FFFF. In the PROMAL manuals, numbers appearing in the text with a $ prefix indicate a hexadecimal number.

You do not use the $ prefix for EXECUTIVE commands, because the EXECUTIVE implicitly expects hex values.

Normally, blanks are treated as separators between arguments. If you wish to specify an argument which contains an embedded blank, the argument must be enclosed in quotes (either " or '). The EXECUTIVE will then treat the entire quoted string as one argument. It will remove the quotes before acting on the argument. Also, the EXECUTIVE normally "folds" all lower case letters in arguments to upper case before acting on the argument. However, if the argument is enclosed in quotes, no conversion takes place. For example:

```
SET 5000 "Now we will learn PROMAL"
```

will treat the entire quoted character string as one argument, and will not convert it to upper case (this command installs the string specified into memory starting at location 5000 hex).

DEVICES

The PROMAL EXECUTIVE (as well as PROMAL programs) can perform input and output to certain devices as well as files. PROMAL devices are named with a single character, as shown in **Table 4** below.

TABLE 4

DEVICE NAMES FOR PROMAL

Name	Meaning
S	The **S**creen. For output only.
K	The **K**eyboard. For input only.
P	The **P**rinter. For output only.
N	The **N**ull device (discards all output). For output only.
W	The **W**orkspace (in-memory file). For input or output.
L	The **L**ibrary (in-memory file). Normally for input.
T	The **T**elephone (modem). For input/output.

Most EXECUTIVE commands can accept one of these device names anywhere a file can be specified. For example:

```
TYPE L
```

will type the contents of the library on the display.

The W device is a simulated file in memory, also called the Workspace. Although the Workspace is small compared to a disk, it is much faster. The Workspace is often used as a place to save a source program you are working on temporarily. The size of the Workspace can be varied on the Commodore 64 by the WS command. Naturally, if you turn off the computer or leave PROMAL, the

contents of the workspace are lost. You can save the Workspace on disk with the COPY command.

The N device simply discards whatever output it receives. This may sound useless but is sometimes useful, as you will shortly see after we discuss I/O redirection.

I/O REDIRECTION

Most EXECUTIVE commands normally output to the screen. However, output may be **redirected** to a file or device by using the redirection operator ">" after the last argument. For example:

```
TYPE MYLETTER.T >P
```

will type the file MYLETTER.T on the printer instead of the screen. Similarly the command

```
FILES >W
```

will output the names of all the files on the disk to the Workspace. You may also redirect output to a file, for example:

```
DUMP 4000 4100 >MEMDUMP.T
```

will output the memory contents to the text file MEMDUMP.T instead of to the screen as it normally would. You can only redirect output to one device at a time.

Many PROMAL programs also allow their output to be redirected in the same manner. Suppose that you had an application program which produced verbose output on the screen each time it ran, and that you wanted to run it without seeing any output. You can do this by redirecting output to the N device.

Some PROMAL programs also allow input redirection. In this case, the program normally accepts input from the keyboard, but can alternatively accept input from a file or device. To redirect input, the input redirection symbol (<) should be used after the last argument. For example, suppose a PROMAL program called INVENTORY normally accepts input from the keyboard and generates a file specified as the first argument. You might tell the program to take its input from another file called APRILINPUT.T instead like this:

```
INVENTORY APRIL.D <APRILINPUT.T
```

The programming techniques for interfacing to the EXECUTIVE are described in the PROMAL LANGUAGE MANUAL, and make it quite simple to support this kind of command.

PROMAL EXECUTIVE COMMAND SUMMARY

On the following pages are descriptions of the individual EXECUTIVE commands. Commands are presented with a syntax definition, a description of the command's function and examples of use. Unless otherwise noted, the commands are available on both Apple and Commodore 64.

BUFFERS — Specify Number of ProDOS Disk Buffers — BUFFERS

AVAILABLE ON APPLE II ONLY.

BUFFERS [Number][HIRES]

The Apple II ProDOS operating system requires that a 1024 byte buffer be allocated in memory for each open file. The BUFFERS command is used to set or display the number of buffers you wish to have set aside. The default is three, which allows three open files at once. Typing BUFFERS without any arguments displays the number of buffers presently assigned. Typing a number after BUFFERS will set the number of buffers specified. Setting the number of buffers will cause all programs to be unloaded, and will affect the memory map. PROMAL allocates the buffers below the available program space.

You also use the argument HIRES (alone or in combination with a number). This will cause PROMAL to unload all programs in memory and reserve space for the hi-resolution graphics "page" from $2000 to $3FFF. This will be necessary before running any program which uses Apple Hi-Res Graphics. You can deallocate the hi-res buffer by typing BUFFERS with a number but without the HIRES argument.

Example 1:

```
BUFFERS
```

will display the current number of 1024 buffers reserved, for example:

```
NUMBER OF BUFFERS = 3
```

Example 2:

```
BUFFERS 1 HIRES
```

allocates one 1024 file buffer plus an 8K byte hi-res screen at $2000. This will of course reduce the size of a PROMAL program which can be loaded.

Notes:

1. You can include a BUFFERS command in a JOB file without harm, even though the buffers for the file may move, so long as there is a buffer available for the job file when the command is completed.

2. Reducing the number of disk buffers below three may adversely affect the operation of the PROMAL COMPILER, since it may need up to three files at once.

3. There is not enough free memory to COMPILE a program after a BUFFERS HIRES command. Therefore when developing a graphics application, remember to switch back to normal mode with a BUFFERS 3 command before compiling.

COLOR -- **Change text and background color** -- **COLOR**

AVAILABLE ON COMMODORE 64 ONLY.

COLOR Number [Bkgndnum]
or
COLOR Colorname [Bkgndcolorname]

The COLOR command is used to change the color used to display text on the screen, and optionally to change the background color. The first argument is the name or number of the desired color for the text, chosen from the following:

0	BLACK	4	PURPLE	8	ORANGE	C	GRAY2
1	WHITE	5	GREEN	9	BROWN	D	LTGREEN
2	RED	6	BLUE	A	LTRED	E	LTBLUE
3	CYAN	7	YELLOW	B	GRAY1	F	GRAY3

The second argument is optional. If specified, it selects the background color. If not specified, the background color is unchanged. Naturally, if you select the same foreground and background color, the text will be invisible (but the computer will still "see" what you type). If you specify a number greater than $F, the EXECUTIVE will force it into the range of 0 to $F by discarding all but the low order 4 bits. Names must be spelled exactly as shown above in order to be recognized (for example, GRAY2 is okay but GRAY 2 is not, nor is GREY2).

Example 1:

```
COLOR PURPLE
```

selects purple characters from this point on.

Example 2:

```
COLOR A 0
```

selects light red characters on a black background.

COPY -- **Copy file or device** -- **COPY**

COPY Filename
or
COPY Source Dest
or
COPY Filename Prefix

The COPY command is used to copy the contents of a file (or device) to another file or device.

COPY for Apple II

If only one argument is given, it must be a file name, not a device name. It may have a prefix or drive designator specified. The destination for the copy of the file is decided as follows: If a prefix was specified and differs from the current prefix, then the file is copied from the specified prefix to the current prefix. If no prefix was specified, the copy is made from the present drive to the "other" drive (1 to 2 or 2 to 1), if it exists and is ready; otherwise, you will be prompted to swap diskettes for a single drive copy.

In the second form, the **Source** and **Dest** may be file names or device names. The copy is made from the Source to the Destination. The destination file name may be different.

In the third form, the first argument must be a file name, not a device name, and the second argument must be a prefix or drive designator. The file will be copied to the specified prefix with the same name.

For all file names, default file extensions will be applied except for arguments in quotes or devices.

Examples for Apple:

If command is...	and...	Then...
COPY MYFILE	Single drive	will copy MYFILE.C to another disk, prompting for disk changes.
COPY MYFILE	2 drives, current prefix on drive 1	copies MYFILE.C from drive 1 to 2.
COPY 1:MYFILE	current prefix is /RAM/ (RAMdisk)	copies MYFILE.C from drive 1 to /RAM/.
COPY MYFILE.T YOUR.T	anything	copies MYFILE.T to YOUR.T on the current prefix.
COPY PROG.S W	anything	copies PROG.S from the current prefix to the Workspace.
COPY 1:PROG.S 2:	anything	copy file PROG.S from drive 1 to drive 2.
COPY MINE.S YOURS	anything	copy file MINE.S to file **YOURS.C** on the current prefix.
COPY L S	anything	copy the L device (library) to the screen.
COPY COMPILE 0:	anything	copy COMPILE.C from the current prefix to the /RAM/ disk
COPY "PRODOS" 2:	anything	copy file PRODOS from the current prefix to drive 2 and **change its name to PRODOS.C.**
COPY "PRODOS" "2:"	anything	copy file PRODOS from the current prefix to drive 2 with the same name.
COPY /MY.LET/JOE.T	current prefix is /TEMP/	copy JOE.T from prefix /MY.LET/ to /TEMP/ with the same name.
COPY 2:PROG.S P	anything	copy file PROG.S on drive 2 to the printer.

The COPY command can only copy one file at a time and does not support wildcards. However, Apple PROMAL has a utility program, **EXTCOPY**, which can copy multiple files using wildcards (* and ?). It's syntax is:

EXTCOPY Pattern Prefix

where Pattern is the desired Filename pattern with wildcards, and Prefix is the destination prefix.

Example:

EXTCOPY *.C 0:

copies all files ending in ".C" from the current prefix to the /RAM disk.

COPY for Commodore 64

If only one argument is given, it must be a file name, not a device name. It may have not have a drive designator specified. You will be prompted to swap diskettes for a single drive copy.

In the second form, the **Source** and **Dest** may be file names or device names. The copy is made from the Source to the Destination. The destination file name may be different. Drive designators (0: or 1:) may be used.

In the third form, the first argument must be a file name, not a device name, and the second argument must be a drive designator. The file will be copied to the specified drive with the same name.

For all file names, default file extensions will be applied and a file type of SEQ will be used, except for arguments in quotes. For names in quotes, no default file extension will be applied, and you may specify a file type explicitly by appending a comma and a letter (S for SEQ, P for PRG, or U for USR) to the name inside the quotes (for example, "Basic Prog,P". When enclosed in quotes, names are case-sensitive.

Examples for Commodore:

If command is...	Then...
COPY COMPILE	will copy COMPILE.C to another disk, prompting for disk changes (single drive copy).
COPY MYFILE YOURFILE	will copy MYFILE.C to YOURFILE.C on the same disk.
COPY SOURCE.S	copy SOURCE.S to another disk, prompting for disk changes (single drive copy)
COPY SOURCE.S W	Copy file SOURCE.S to the Workspace.
COPY W S	Copy the workspace to the screen.
COPY MYFILE 1:	Copy file MYFILE.C from drive 0 to drive 1.
COPY 1:TEST.S 0:OLDTEST	Copy file TEST.S from drive 1 to drive 0 with a name change to **OLDTEST.C**
COPY 1:TEST.S 0:OLDTEST.S	Copy file TEST.S from drive 1 to drive 0 with a name change to OLDTEST.S
COPY TEST.L P	Copy file TEST.L to the printer.
COPY "PROMAL,P"	Copy file PROMAL of type PRG to another disk, prompting for disk changes (1 drive)

COPY for both computers

Single drive copies can be made for files up to 64K bytes long. Large files may require several disk swaps. Other copies are limited only by the available disk or device space.

If the file already exists on the destination diskette, you will be prompted:

```
FILE XXXXX EXISTS.
APPEND, REPLACE, OR CANCEL (A/R/C)? _
```

Press the A key to append onto the end of the existing file, R to replace the existing file, or C to cancel the command.

Notes:

1. You will not be able to append the Workspace; copying to it discards any previous contents.
2. Do not attempt to execute a COPY command from a JOB file on disk in a single drive system if the copy will require disk swaps, because PROMAL will attempt to read the JOB file while your destination disk is in the drive. You may execute such a job file by first copying the job file to the Workspace and then using a JOB W command.
3. See **Appendix N** for Commodore dual drive installation instructions.
4. Do not attempt to copy Commodore 64 relative files.
5. On the Apple, attempting to replace a locked file will give a WRITE PROTECTED error message.
6. Attempting to append a locked file or file on a write protected disk will produce a DISK ERROR message.
7. Copying a file to the W device which is larger than the workspace will result in a DISK/DEVICE FULL error with the rest of the file not copied.
8. On the Apple, copying a file from one drive to another when both drives have the same volume name will give an ILLEGAL FILE/DEVICE NAME error.
9. The copy command uses the free memory space for a copy buffer. Therefore you can increase the efficiency of copy operations (especially for single drive copies) by UNLOADing memory before copying.

CS -- Clear screen and home cursor -- CS

CS

The CS command clears the screen and moves the cursor to the home position. It does not have any arguments.

Example:

```
CS
```

DATE -- **Display and set date** -- **DATE**

DATE

The DATE command prompts for the current date. It is the same command that is executed automatically when the EXECUTIVE signs on. If you have already entered the date, it will display the current date and then ask you for the desired date. You may enter the date or simply reply with RETURN to keep the present date. The PROMAL compiler uses this date to "stamp" all compiled programs with their creation date. This creation date is displayed by the MAP command when programs are in memory, or by the SIZE command if they are on disk.

Example:

DATE

```
    Today is 9/30/86
    Please enter today's date: _
```

Note:

1. On the Apple IIe, the date will be updated automatically on systems with a Thunderclock or equivalent card when the DATE command is executed, without any prompt or user action.
2. On the Commodore 64, DATE is not a built-in command, but a separate program, which is automatically unloaded after it executes. Therefore you will need to have the file DATE.C on your boot disk to have the DATE command run.
3. If no BOOTSCRIPT.J is present on the boot disk, the PROMAL EXECUTIVE will execute DATE automatically during bootup. If a BOOTSCRIPT.J file is present, DATE will not be automatically executed, so you should have a DATE command in your BOOTSCRIPT.J file unless you don't want to set the date (see JOB for more information).

DELETE -- **Delete file from directory** -- **DELETE**

DELETE Filename [...]

The DELETE command removes one or more specified files from the disk directory. The file name(s) to be deleted should be specified as the argument(s). Wildcards are **not** permitted in the Filename. You may not delete a device, only a file name. However, typing DELETE W will clear the workspace. Attempting to DELETE a file on a write-protected disk (or a locked file on an Apple II) will produce an error message.

CAUTION: There is no prompt for verification before the file is deleted, nor is there any way to "un-delete" a file, so use DELETE with caution. You should maintain backups of all important files (for any system, not just PROMAL).

Example:

```
    DELETE MYPROG
```

deletes the file MYPROG.C from the diskette.

You may delete files with no file extensions by specifying the file name in quotes, for example:

```
DELETE "PRODOS"
```

For Non-PROMAL files on the **Commodore 64**, file names enclosed in quotes are case sensitive. For example **delete "promal"** will not delete the file PROMAL but **delete "PROMAL"** will. PRG and USR type files may be deleted.

Notes:

1. DELETE only removes a file from the disk directory. If the deleted file is a program which is also in memory, it is not removed from memory and can still be executed. The UNLOAD command removes the memory-resident program.
2. On the Apple II, you may delete subdirectories, provided they are empty. Remember to enclose the name in quotes.

DISKCMD — Access disk command/error channel — DISKCMD

AVAILABLE ONLY ON THE COMMODORE 64

DISKCMD [Command]

The DISKCMD can be used to send a command to the Commodore disk command channel, and to display the status message from the disk error channel. For normal operations, this command is never needed, since PROMAL and the EXECUTIVE normally handle all command/error processing transparently. You should use this command only for special circumstances requiring special disk commands. If no argument is given, the current disk status will be displayed, for example:

```
00, OK,00,00
```

If the argument is specified, it should be a Commodore disk command enclosed in quotes.

Example:

The following command will **immediately** format the disk currently in the drive, erasing everything currently on the disk:

```
DISKCMD "NO:WORKDISK1,K6"
```

CAUTION: Be very sure you really want to format the disk before typing this command, there is no chance to change your mind!

The example above formats drive 0 (device 8) with a disk name of WORKDISK1 and a disk ID of K6. You should always use a different two letter ID for every disk you format.

A good use of DISKCMD is to issue a "I0" command if you have changed diskettes and think that the new and old diskette may have the same ID. Refer to the Commodore 1541 or 1571 manual for more information on commands which may be issued to the command channel.

DYNO — Enable or disable double speed disk — DYNO

AVAILABLE ONLY ON THE COMMODORE 64

DYNO [Onoff]

PROMAL for the Commodore 64 has DYNODISK, a built-in software package which effectively **doubles the read speed** from Commodore 1541 or 1571 disk drives. Needless to say, this is a tremendous asset, since the Commodore disk is notoriously slow. DYNODISK works with **all** files, not just programs. The DYNO command with no arguments specified will display the current DYNODISK status, either ON or OFF. The optional command **Onoff** must be either the word ON or OFF, and turns the DYNODISK feature on or off.

Examples:

```
DYNO
```

will display the current status, as either ON or OFF.

```
DYNO OFF
```

disables DYNODISK for further operations until turned back on. With DYNO OFF, the 1541 will operate at normal speed.

Notes:

1. DYNODISK cannot be used with MSD drives or other drives which are not 100% compatible with the 1541. You can permanently disable dynodisk by setting the byte at $0DE2 non-zero. Do not enable DYNODISK with Skyles FLASH or with other commercial disk speed up cartridges or software. Most other commercial disk speedup products do not work with PROMAL at all because they use some of the same memory needed by PROMAL.

2. Like any 1541 speedup package, DYNODISK does have its drawbacks. **If you have a printer or other device on the serial bus, it must be turned off while DYNODISK is ON.** Failure to observe this rule may hang up the system, requiring a re-boot of PROMAL. If you have a printer with an interface cartridge, it must also be off while DYNODISK is on. Only a single drive is supported. Having DYNODISK on reduces the number of buffers which are available inside the 1541 drive. Therefore you may be able to have fewer open files while DYNODISK is enabled. DYNODISK disables interrupts temporarily while it is reading from disk. Therefore you will not be able to use CTRL-STOP to abort a program while it is reading the disk with DYNODISK on.

3. DYNODISK only doubles the transfer rate of information from the disk drive to the computer. It does not improve the access time or time needed to open a file. Therefore you will observe the biggest speed reductions when reading large files, and little or no improvement on small files. For reads of very small amounts of information, having DYNODISK enabled may actually take slightly longer due to a small setup overhead required. DYNODISK only affects reading speed, not writing speed. See GETBLKF in the LIBRARY manual for programming considerations using DYNODISK.

4. On a 1571 drive, do not send the disk commands to enable the double-sided mode or any of the faster, non-C64-compatible disk modes, with or without DYNO on.

DUMP -- Display memory in Hex and ASCII -- DUMP

DUMP Address [ToAddress]

The DUMP command is used to display a region of memory on the screen in hexadecimal and in ASCII characters. The first argument is the desired starting address. The second, optional argument is an ending address. If no second address is specified, eight bytes will be displayed.

Examples:

```
DUMP 10A0
```

will display eight bytes of memory beginning at $10A0. The display will appear similar to this:

```
10A0   50 0C 4C D6 29 4B 00 52   P.L.)K.R
```

The starting address is shown in the leftmost column. The next eight 2-digit groups show the hex values of $10A0 through $10A7. The rightmost column shows eight characters with the ASCII character equivalent of each of the bytes. Bytes which don't have a printable ASCII character (including blanks) are displayed as "." instead.

```
DUMP 4000 4100 >P
```

will dump the contents of $4000 through $4100 to the printer.

Note: When an end address is specified, a complete line will always be displayed even if it "overshoots" the final address. Therefore the above example will actually display $4000 through $4107, since $4100 will fall in the first position of the last line of the output.

EDIT **-- Invoke PROMAL Editor --** **EDIT**

EDIT [Filename]

The EDIT command is used to invoke the PROMAL full-screen EDITOR. The optional argument is the name of the file to be edited. If no file is specified, the Workspace will be used. If a file is specified and exists, it will be edited. Otherwise, it will be assumed to be a new file to be generated.

The EDITOR is described separately in the EDITOR section of this manual.

Example:

```
EDIT MYPROG.S
```

edits the file MYPROG.S

Note for **Commodore 64** systems: Normally the EDITOR is always resident in memory and will start immediately when EDIT is typed. However, it is possible to unload the EDITOR from memory by using the "B" option when running the COMPILER to free up extra room for the symbol table (see the COMPILER section of this manual). In this case the EDITOR will automatically be reloaded from disk when needed. The Editor will also be reloaded from disk if the EDITOR program has been corrupted (for example, by an errant user program poking around in memory). You may EDIT larger files by UNLOADing programs or clearing or reducing the Workspace before starting the Editor.

FILES **-- Display diskette directory --** **FILES**

FILES
or
FILES Pattern* **COMMODORE 64 ONLY FOR THIS FORM**
or
FILES Subdirectory **APPLE II ONLY FOR THIS FORM**

The FILES command is used to display the names of files on disk.

FILES for the Apple II:

If no argument is given, all files on the current prefix will be displayed, in four columns. If an argument is given, it should be the desired prefix. No wildcards are supported.

Examples:

```
FILES                     Displays all filenames in current prefix
FILES 2:                  Displays all file names on drive 2
FILES /USER.DISK/TEMP/    Displays all files in specified directory
FILES 0: >P               Prints the names of all files on the /RAM disk.
```

Note:

1. Redirecting FILES to the printer prints one file name per line.

For a more detailed listing of the directory with file sizes, dates, and other information on the Apple, use the **EXTDIR** command. The EXTDIR command is not a built-in command, but a compiled PROMAL program. It requires an argument specifying a pattern to match. The wild cards are * and ?.

Examples for Apple:

EXTDIR *	Displays all files on the current prefix
EXTDIR 2:	Displays all files on drive 2
EXTDIR /TEMP/T*.C	Displays all files starting with T with a .C extension in the /TEMP/ directory.

You will need file EXTDIR.C in order to use the EXTDIR command.

FILES for the Commodore 64:

The files will be listed on the screen in the same format as for BASIC, including the file sizes. Wild cards may be used to display only selected files. The wildcard characters operate exactly as described in the Commodore 1541 Disk Manual. If no Pattern is specified, all the files on the disk will be displayed. The size of each file is measured in Commodore Blocks, which are 256 bytes each. The number displayed is in decimal, not hex. Displaying files does not affect programs in memory. The wildcards allowed are * and ?. The * character matches any string, and the ? character matches any single character.

Examples for Commodore 64:

FILES	Displays all files on drive 0
FILES PR*	Displays all files starting with "PR"
FILES 1:	Displays all files on drive 1
FILES >P	Prints the names of all files

Note:

1. You might suppose it would be possible to display all file names with the extension ".S" by using the "FILES *.S" command. Unfortunately, the Commodore ROMs in the disk drive do not interpret the * wildcard this way. Instead, the *.S pattern is interpreted as matching all files on the disk, no matter what the names are. Therefore the * wildcard is only useful for matching names with a common prefix which differ only in the suffix. The * wildcard cannot be used to represent a common suffix.

2. Redirecting FILES to the printer prints one file name per line.

FILL -- **Fill memory area with constant** -- **FILL**

FILL From To Data

The FILL command is used to fill a region of memory with a constant. The first argument, From, is the starting address. The second argument, To, is the final address to fill. The third argument, Data, is the byte to fill with. All arguments are normally specified in hexadecimal. However, the Data argument may be specified as a character in single quotes (') if desired.

CAUTION: If you want to experiment with the FILL command, specify an unused area of memory (shown as "FREE SPACE" by the MAP command). Indiscriminate use of the FILL command could overwrite important programs (such as PROMAL itself) in memory.

Examples:

```
FILL 4800 4923 0
```

fills $4800 through $4923 with $00.

```
FILL 4210 4310 ' '
```

fills $4210 through $4310 with ASCII blanks ($20).

FKEY -- **Display or change function keys** -- **FKEY**

FKEY [Keynumber String]

The FKEY command is used to display or change the meaning of the function keys, F1 through F8. If no arguments are given, the current function key definitions for all function keys will be displayed. If the optional arguments are specified, then the first argument is the desired function key number to change, 1 through 8. The second argument is the desired character string to be substituted when the function key is pressed. The String may be up to 31 characters long. If it contains blanks, then it should be enclosed in quotes. Only normal, printable characters can be included in the String. Control characters (such as RETURN) cannot be embedded in the String.

Examples:

```
FKEY
```

will display the current function key definitions. A typical display would be:

Commodore 64	Apple II
F1 = EDIT	F1 = EDIT
F2 = DUMP	F2 = PREFIX *
F3 = COMPILE	F3 = COMPILE
F4 = GET	F4 = GET
F5 = FILES	F5 = FILES
F6 = MAP	F6 = EXTDIR *
F7 = HELP	F7 = HELP
F8 = COPY	F8 = COPY

This is the default list of function key definitions at power-up.

```
FKEY 2 MYPROGRAM
```

changes function key F2 to generate "MYPROGRAM" when it is pressed.

```
FKEY 5 "COMPILE MYPROGRAM O=NEWVERSION"
```

changes F5 to be "COMPILE MYPROGRAM O=NEWVERSION".

If you wish to have certain function keys automatically defined when you "boot up" PROMAL, you may do so by simply including the appropriate FKEY commands in the BOOTSCRIPT.J file on your working diskette.

GET -- Load program from diskette -- GET

GET Progname

The GET command is used to load a PROMAL or machine language program into memory without executing it. The argument, **Program,** is the desired file name to be loaded. Normally it is a legal PROMAL file name, written **without** quotes. Only compiled PROMAL programs (or relocatable machine language programs in the PROMAL format as described in Appendix I) can be loaded using this form. An attempt to GET a file of another type will result in an error message. After the program is loaded, it will appear on the MAP display, and can be executed immediately by typing its name.

Alternatively, the Program can be the name of any non-relocatable Machine Language program, enclosed in quotes ("). In this case, the named machine language program will be loaded into memory at the address from which it was saved. No checking is done for overlapping of other programs, nor is space allocated for the program in the MAP. See Chapter 6, function MLGET in the LIBRARY MANUAL, and Appendix I for more details.

Examples:

```
GET SORT
```

loads the PROMAL program SORT.C into memory (because .C is the default extension). The actual load location will be determined by the LOADer, and can be displayed using the MAP command.

```
GET "MLSUBS"
```

loads the non-relocatable machine language file called "MLSUBS" into memory at its formerly saved address. It will not show up in the MAP display.

Notes:

1. PROMAL allows several programs to be resident in memory at once, subject to the amount of free memory left. The LOADer will relocate the PROMAL program to an available location. If there is not enough room left, the LOADer will unload programs one at a time, beginning with the last-loaded program, until there is enough room. When the program is loaded, the LOADer also computes and saves a "checksum" of the program image in memory. When you subsequently execute the program, the LOADer will re-compute the checksum and compare it to the saved value. If the two values differ, it indicates that the program in memory has been corrupted (by another program), and so the LOADer will re-load the program from disk before executing it.

2. On the **Apple II,** most machine language programs load initially at $2000 and then "relocate" themselves to their final location. In order to load such a program with the GET command, you need to do a BUFFERS HIRES command first to free the space from $2000 to $3FFF. You also need to insure that the program will not have any other memory "collisions" with PROMAL, including its "relocated" destination, zero page usage, etc.

3. On the **Apple II,** if you GET a PROMAL program, and no error is issued, but when you use the MAP command it does not appear, it means that your program was successfully loaded, but was immediately unloaded when the EXECUTIVE was swapped back in. You will need to UNLOAD. If the symptom persists, then the program is too large to fit in memory at the same time as the EXECUTIVE; you can execute it by typing its name, but you can't GET it from the EXECUTIVE. The COMPILE program exhibits this characteristic.

GO -- Execute Machine Language program -- GO

GO Address

The GO command is used to execute a machine language program already in memory (not a PROMAL program). The argument specifies the starting address for execution in hex. The GO command cannot be used to execute a PROMAL program.

Example:

```
GO FF81
```

will execute the machine language program at address $FF81. On the Commodore 64, this address is the Commodore 64 Kernal routine "CINT" which re-initializes the screen and video chip.

Notes:

The machine language routine is entered via a 6502 JSR instruction. If the machine language program exits with an RTS instruction, it will return to the PROMAL EXECUTIVE in the normal way. If a machine language BRK instruction is encountered, control will be returned to the EXECUTIVE with an error message similar to:

```
*** RUNTIME ERROR: M/L BRK HIT
P       A   X   Y   F   S
6B01    A3  B2  11  32  F6
AT $5000
*** PROGRAM ABORTED.
```

This display gives the contents of the 6502 registers at the time of the breakpoint and the address of the PROMAL instruction which called the machine language routine. If a GO command was used to enter the program, it will show the starting address instead.

HELP — **Display help screen** — **HELP**

HELP

The HELP command displays a single screen of information showing some of the control keys used for editing and a list of the most commonly-needed EXECUTIVE commands.

Example:

```
HELP
```

on the Commodore 64 will display a screen similar to:

```
               PROMAL HELP
CTRL-                        CTRL-
A Upper Alpha On/Off           Delete Char.
B Recall Prior Line          [ CRSR to start
K Clear to End Line          Y CRSR to End
              Partial Command Summary
COLOR [Colorname [Background]]
COMPILE [File [L[=List]][O=Object][B]]
COPY File [Dest.]
DELETE File                 Function Keys
DUMP From [To]               F1 = EDIT
EDIT [File]                  F2 = DUMP
FILES [Pattern*]             F3 = COMPILE
FILL From To Value           F4 = GET
FKEY [Number String]         F5 = FILES
GET Commandfile              F6 = MAP
JOB File.J                   F7 = HELP
MAP                          F8 = COPY
RENAME File Newname
SET Addr. Val [Val...]
Type File
Unload [Command]
```

The Apple II help screen is somewhat different (see **MEET PROMAL!**).

Note: Any function key definitions longer than 8 characters will have only the first 8 characters displayed on the HELP menu. The FKEY command will print the entire definition.

JOB -- Execute commands in job file -- JOB

JOB File.J

The JOB command allows the EXECUTIVE to accept a list of commands from a file on disk instead of the keyboard. This is sometimes called a "batch" capability. Normally this file of commands is prepared using the PROMAL EDITOR. The EXECUTIVE will read commands from the job file until end of file is reached or an error is encountered. Commands should appear in the job file just as they would be typed from the keyboard. Both built-in and user-defined programs may be executed from the job-file "script".

Example:

```
JOB SCRIPT1.J
```

will cause the EXECUTIVE to read and execute the list of commands on the file SCRIPT1.J. For example, this file might contain:

```
FILL 4000 4700 0   ; zero memory segment
GET "MYMLSUBS"     ; load special machine language subroutines
MAINPROG           ; load & run my PROMAL program
DELETE TEMPJUNK    ; get rid of unneeded scratch file
PROG2 >P           ; run my second program, redirect output to printer
```

The EXECUTIVE will attempt to execute all five of these commands before accepting more commands from the keyboard.

When PROMAL is first booted up, it looks for a special JOB file on your working disk called **BOOTSCRIPT.J.** If it finds this file, it will execute the commands on it before accepting commands from the keyboard. You can EDIT the BOOTSCRIPT.J file to do whatever you want. For example, you may want to change the Commodore screen colors with the COLOR command, change the function key definitions with the FKEY command, or execute a certain program automatically when you start the system.

If no BOOTSCRIPT.J is found, the EXECUTIVE will execute an FKEY command and a DATE command by default, to display the function keys and prompt for the date. If you **do** have a BOOTSCRIPT.J file and want the prompt for the date, you should include the DATE command (and the FKEY command if you want) in the script. For the Commodore 64, you will need to have the file DATE.C on disk.

PROMAL supports another feature which greatly enhances the power of JOB files. You can pass arguments (called macros) to a JOB file which will be substituted for "placeholders" in the script. These "placeholders" consist of a \ character (or a "pounds sterling" character on the C-64) followed immediately by a single digit indicating which argument should be substituted. For example \1 will be replaced by the first argument, \2 by the second argument, etc. The first argument is specified after the file name on the JOB command. An example may clarify all this:

Suppose you prepared a file called DO.J which looked like this:

```
EDIT \1.S
DELETE \1.1
COMPILE \1 L=\1.L
UNLOAD \1
\1 TESTDATA.D
```

Now suppose you use the command:

```
JOB DO.J MYPROG
```

The result will be that PROMAL will execute the following commands:

```
EDIT MYPROG.S
DELETE MYPROG.L
COMPILE MYPROG L=MYPROG.L
ULOAD MYPROG
MYPROG TESTDATA.D
```

Each occurrence of \1 was replaced with the first argument (after the file name DO.J) as the commands were read.

Notes:

1. Comments may be included in the Job file, preceded by ";".
2. The ".J" must be explicitly specified in the JOB command.
3. A JOB command may not appear in a JOB file script, except as the last command in the file. You can make a job file that "loops" by making the last command a JOB command with the same filename.
4. A runtime error or any program which executes the library procedure ABORT will terminate the job file and return control to the keyboard.
5. You may execute a JOB file in the Workspace by typing JOB W. Of course you shouldn't use the Workspace for anything else at the same time.
6. The \ character is the "pounds sterling" (£) key on the Commodore 64.
7. Don't put any command in the JOB file that will require swapping the disk the JOB file is being read from (such as a one-drive copy).
8. The JOB file uses one disk buffer, so you may not be able to execute some file-intensive command from a JOB file which works fine when executed directly from the keyboard. Copying the JOB file to the workspace and executing JOB W may alleviate this problem.
9. See the PAUSE command for a useful command in JOB files.

LOCK -- **Set write-protect for file** -- **LOCK**

AVAILABLE ON APPLE II ONLY

LOCK Filename [...]

The LOCK command sets the write-protect lock on the specified file names. Locked files cannot be written to, renamed, or deleted until they are unlocked.

Example:

```
LOCK COMPILE
```

will write-protect the file COMPILE.C.

Note: Locking a file only provides protection against inadvertent deletion or modification. It does not protect files from hostile users, other software, or from erasure by formatting a disk, nor does it prevent copying or use of the file. See the ProDOS Technical Reference Manual for more details.

MACRO -- **Define Macro String** -- **MACRO**

MACRO String [String...]

The MACRO command allows you to define macro substitution strings (in the same manner as is described for the JOB command), which can be used interactively instead of in a JOB file. A macro allows a string to be substituted for a two-character abbreviation in an EXECUTIVE command when it is processed. The two character abbreviation consists of the backslash character, **** (or "pounds sterling" key on the Commodore 64) followed by a number from 1 to 8, indicating which string is to be substituted. The MACRO command defines the substitution strings to be used in subsequent commands. The first argument will replace \1, the second argument \2, etc.

Example:

```
MACRO /DEVEL/MYPROG
```

After this command is issued, then:

```
COMPILE \1
```

will be processed by the EXECUTIVE exactly as though you had typed:

```
COMPILE /DEVEL/MYPROG
```

Example 2:

```
MACRO 2:PROCESSDATA " 10 100 200 300"
```

If you then type:

\1\2

this will be processed by the EXECUTIVE as:

2:PROCESSDATA 10 100 200 300

Notes:

1. The MACRO command is particularly handy for defining frequently needed volume or path names on the Apple.
2. Macros are independent of function key definitions and may be used inside function key definitions.
3. Macro strings may be any length which will fit the same line as the MACRO command. They may not be nested.
4. A JOB command or another MACRO command redefines all macros.

MAP **-- Display current memory map --** **MAP**

MAP

The MAP command is used to display the load addresses of any programs in memory, and a summary of how memory is currently allocated. The actual format of the MAP display varies somewhat depending on the memory organization of the computer.

Example:

```
MAP
```

will display the current memory map. If no programs have been loaded yet, the map should appear similar to this (the actual addresses may differ):

<u>Apple II</u>

```
OBJECT PROGRAMS   $2900-      (0)
FREE SPACE        $2900-8DFF (25856)
SHARED VARIABLES  $8E00-      (0)
EXEC./EDIT SPACE  $6100-8DF8 (11520)
TOTAL SPACE       $2900-8DFF (25856)

ACTIVE WORKSPACE  $1200-      (0)
FREE WORKSPACE    $1200-5AFF (16688)
```

<u>Commodore 64</u>

```
OBJECT PROGRAMS   $4F00-      (0)
FREE SPACE        $4F00-98FF (18944)
ACTIVE WORKSPACE  $9900-      (0)
FREE WORKSPACE    $9900-A0FF (2048)
SHARED VARIABLES  $A100-      (0)
EXEC./EDIT SPACE  $A200-CFFF (11776)
TOTAL SPACE       $4F00-CFFF (33024)
```

The numbers in the FROM-TO column are in hexadecimal. The numbers in parentheses are the size in decimal. The meaning of the displayed lines is as follows:

OBJECT PROGRAMS	Indicates the total space allocated for all programs currently loaded in memory (excluding shared variables).
FREE SPACE	Indicates the total space presently available for additional programs and variables. This space can also be used to increase the Commodore Workspace (with the WS command). The available space can be increased by UNLOADing programs, reducing the Workspace, or using a NOREAL command. For the Commodore 64, programs may additionally use the EDITor space if needed (described in Chapter 8 of the PROMAL LANGUAGE MANUAL).
ACTIVE WORKSPACE	Indicates how much of the Workspace is currently in use. A WS CLEAR command will set this to 0. For the Apple II, the Workspace is maintained in auxiliary (banked) memory. On the Commodore 64, the Workspace may move as programs are loaded (the contents are maintained).
FREE WORKSPACE	Indicates the size of the unused portion of the Workspace. The WS command can be used to alter this value.
SHARED VARIABLES	Indicates the space allocated for un-initialized global variables (arrays and reals). These variables are allocated at the top of available memory.
EXEC./EDIT SPACE	This is the space occupied by the EXECUTIVE or the EDITOR, or by a users program or variables if needed. The EXECUTIVE and EDITOR share the same space in memory. Only one or the other (or neither) is present at any given time. For the Commodore 64, when a program is executed, the EXECUTIVE is copied to the RAM under the ROMs and the EDITor (which was under the ROMs) is swapped into the vacated space. On exit, they swap again. For the Apple II, copies of the EXECUTIVE and EDITOR are kept in the auxiliary 64K RAM bank and are copied into the EXEC./EDIT space as needed. See Appendix G and Chapter 8 of the PROMAL LANGUAGE MANUAL for more information.
TOTAL SPACE	This is the maximum amount of space presently allocatable for PROMAL programs and variables, if all programs are unloaded, (and, for the Commodore, if the Workspace is cleared and the EDITor's space is used, as described in Chapter 8 of the PROMAL LANGUAGE MANUAL).

After several programs have been run, the MAP command might produce a display that looked more like this:

Apple II

```
FIND        (PRO.)   5/ 2/85 CHKSUM 8171
  CODE $2900-2AFF, VARIABLES $8D00-8DFF
CALC        (PRO.)   8/ 5/85 CHKSUM 8FAF
  CODE $2B00-30FF, VARIABLES $8C00-8DFF

OBJECT PROGRAMS  $2900-30FF (2048)
FREE SPACE       $3100-8BFF (23296)
SHARED VARIABLES $8C00-8DFF (512)
EXEC./EDIT SPACE $6100-8DFF (11520)
TOTAL SPACE      $2900-8DFF (25856)

ACTIVE WORKSPACE $1200-     (0)
FREE WORKSPACE   $1200-5AFF (18688)
```

Commodore 64

```
FIND        (PRO.)   1/ 6/85 CHKSUM B5E7
  CODE $4F00-50FF, VARIABLES $A000-A0FF
CALC         (PRO.)  5/24/85 CHKSUM AE67
  CODE $5100-56FF, VARIABLES $9F00-A0FF

OBJECT PROGRAMS  $4F00-56FF (2048)
FREE SPACE       $5700-96FF (16384)
ACTIVE WORKSPACE $9700-
FREE WORKSPACE   $9700-9EFF (2048)
SHARED VARIABLES $9F00-A0FF (512)
EXEC./EDIT SPACE $A200-CFFF (11776)
TOTAL SPACE      $4F00-CFFF (33024)
```

This display shows two PROMAL programs present in memory, FIND and CALC. The line starting with the name of the program shows that it is a PROMAL program, the date it was compiled, and the checksum which the EXECUTIVE uses to verify the integrity of the program before execution (described in the GET command).

The line beginning with "CODE" shows the starting and ending address of the executable code, and the address range of the variables. It is normal for programs to have overlapping variables allocated, which is why this area is called "shared variables" in the summary. However, the variables may also be allocated immediately after the code portion, if the program had the keyword OWN on its PROGRAM line when it was compiled (discussed in the PROMAL LANGUAGE MANUAL).

Memory is always allocated in pages of exactly $100 bytes each (256 decimal). This is why all the address ranges start with $xx00 and end with xxFF. Therefore even if a program is only a few bytes long, it will still be allocated a whole page. See Chapter 8 of the LANGUAGE MANUAL for further information. The SIZE command can be used to determine the true size of a program to the exact byte.

For the Commodore 64, notice that the Workspace has moved slightly to make room for the new shared variables. This will not affect the content of the Workspace, which is maintained by the LOADer.

Notes:

1. Memory areas not shown on the MAP display are used by the PROMAL system. A detailed memory map for PROMAL system usage is shown in **Appendix G.** The EDITOR and COMPILER will use all of the area indicated as FREE SPACE for buffers when they run. In addition, they may use the Commodore Workspace if it is completely empty (see the EDITOR section of this MANUAL for a full discussion of when the EDITOR uses the Workspace). The Commodore COMPILER also has an option of using more memory for its symbol table (see the COMPILER section of this manual).

2. If you GET the Apple compiler, it will not show up in the memory map. This occurs because the compiler will be immediately unloaded from memory after it is loaded to make room for the EXECUTIVE, which overlaps it. This is of no particular consequence, except that you must have a copy of the compiler on disk if you want to run it.

NEWDIR -- Create a new sub-directory -- NEWDIR

AVAILABLE ON APPLE II ONLY

NEWDIR Name

The NEWDIR command is used to create a new subdirectory called Name. The name should be a legal ProDOS sub-directory name. A "/" will be added to the end of the name if one is not specified. If the name does not **start** with a "/" character, then the current specified name will be appended to the current prefix.

Example:

```
NEWDIR MUSIC/      ; Creates new subdirectory /MUSIC/ on current prefix.
```

NOREAL -- Remove real number routines -- NOREAL

NOREAL

The NOREAL command is not normally needed, but allows you to discard all routines from the PROMAL runtime software and resident library which support the processing of REAL (floating point) data. This adds approximately 2.5 K bytes of additional available space for editing, compilations, or other programs. However, you will not be able to successfully execute any program which makes use of REAL data (the EDITOR and EXECUTIVE do not; the COMPILER only uses REALs if the program it is compiling uses REALS).

When the NOREAL command is executed, the EXECUTIVE will first UNLOAD **all programs from memory.** It will then move the bottom of available memory down by about 2.5K. You may then resume normal operations.

If you normally have no need for floating point capabilities (many application areas such as games, music, graphics, and text editing normally don't need REAL data), you may wish to include the NOREAL command in your BOOTSCRIPT.J file (described in the JOB command, above) to automatically free up more space when you boot up.

Once NOREAL has been executed, the only way to restore the REAL processing is to reboot the system.

If you get either of these two messages, it indicates that you may have tried to execute a program which uses REALs after you have discarded REAL processing:

```
*** RUNTIME ERROR: ILLEGAL OPCODE
AT xxxx

*** RUNTIME ERROR: REQ'D PROGRAM NOT LOADED
AT xxxx
```

PAUSE **-- Pause in JOB file --** **PAUSE**

PAUSE ["Message"]

The PAUSE command is normally only used in JOB files. When executed, it displays the text of a message enclosed in quotes and then waits for a carriage return from the keyboard. The text of the message must be enclosed in double quotes.

Example:

PAUSE "PRESS RETURN TO CONTINUE."

When executed in a JOB file, this will pause and display "PRESS RETURN TO CONTINUE" on the screen. If you press RETURN, the job will continue with the next command. Alternatively, you could press CTRL-STOP (Commodore) or CTRL-C (Apple) to abort the job at this point in the JOB file execution.

PREFIX -- Set or view default pathname -- **PREFIX**

AVAILABLE ON APPLE II ONLY

PREFIX [Pathname]
  or
PREFIX * [Slot Drive]

The PREFIX command is used to set or show the current prefix (path name) which will be used for subsequent file references. The PREFIX command without any arguments displays the current path name. If a name is specified as an argument, it should be a legal path name (starting and ending with / characters). This form is most often used to select a different diskette (ProDOS Volume) for subsequent action. The form PREFIX * will change the pathname to whatever the volume name is for the diskette in the drive which booted up PROMAL, and display this name. This form is most often used after changing diskettes.

Examples:

```
PREFIX              will display the current path name, for example
PREFIX *            updates the current prefix to the volume in drive 1
PREFIX 2:           sets the current prefix to the volume in drive 2
PREFIX /WORK1/      sets the current prefix to /WORK1/
PREFIX * 6 1        sets the current prefix to slot 6 drive 1
```

If ProDOS can not find a diskette with the specified volume name, a DEVICE NOT READY error will be given. You can use the PREFIX * command to determine the volume name of an unknown diskette. Volume names are initially assigned when the diskette is formatted.

IMPORTANT: You should always execute a PREFIX * command after changing diskettes. Otherwise, ProDOS will not be able to find your commands or files, because the current prefix will still be the removed volume name.

QUIT -- Quit PROMAL -- **QUIT**

QUIT

The QUIT command exits from the PROMAL system. The workspace will be lost.

For the **Commodore 64**: QUIT resets the computer back to its power-on status in BASIC. You may **not** re-enter PROMAL except by re-loading it from disk.

For the **Apple II:** QUIT will change to 40 column mode and prompt for the name of a "prefix". Enter the name of the desired system to run, which must end in ".SYSTEM". For example, to run BASIC, you might enter /WORK1/BASIC.SYSTEM.

Example:

```
QUIT
```

It is not necessary to QUIT before turning off your computer. You may simply remove the diskette and power off the computer in the normal manner, provided the disk is not being accessed.

RENAME **-- Rename a file --** **RENAME**

RENAME Oldfile Newfile

The RENAME command is used to rename a file on disk. The first argument is the existing file name and the second is the new file name desired.

Examples:

```
RENAME MYFILE YOURFILE
```

changes the file named MYFILE.C to YOURFILE.C.

```
RENAME DATA1.D DATA1.T
```

changes the extension of the file.

For the **Commodore 64**: Non-PROMAL files may be renamed by enclosing the file name (case sensitive) in quotes, for example:

```
RENAME "BASIC PROG" "OLD_PROG"
```

For the **Apple II**: Any path name can be specified for the old file. If you wish the new name to not have a file extension, enclose it in quotes. File names without extensions should be enclosed in quotes. You may rename a directory (in quotes). Remember to change the PREFIX after you do. You can rename a volume name (in quotes, no trailing '/').

SET **-- Set memory address to value --** **SET**

SET Address Value [...]

The SET command is used to install values in memory. The first argument is the starting address, in hex. The second (and optionally additional) arguments specify the values to be installed into memory in sequence. Each **Value** argument can be:

(1). A byte specified in hexadecimal
(2). A word greater than $00FF specified in hexadecimal
(3). A string to be terminated by a 00 byte, enclosed in double quotes (")
(4). A string enclosed in single quotes (')

The difference between (3) and (4) above is that if the string is enclosed in double quotes ("), a 00 byte terminator will be installed automatically after the last byte of the string, but if it is enclosed in single quotes ('), no 00 byte is added.

CAUTION: Indiscriminate use of the SET command may overwrite important programs or data in memory (such as PROMAL itself). Before experimenting with the SET command, you should pick a "safe" location, such as in the FREE SPACE displayed by the MAP command.

Example:

```
SET 54B0 4B
```

sets the byte at location $54B0 to $4B.

```
SET 5700 A921 0 143
```

sets location $5700 to $21, $5701 to $A9, $5702 to $00, $5703 to $43, and $5704 to $01 (note: word values are stored in the standard 6502 processor order with the low-order byte first and the high-order byte at the next address).

```
SET 4B10 'Hello'
```

installs five bytes starting at $4B10 (lower case letters are not changed to upper case).

```
SET 4B10 "Bye now"
```

installs the specified string starting at $4B10, and adds a $00 byte after the last character installed in memory (the $00 byte is used to terminate a string as defined by the PROMAL LANGUAGE MANUAL).

SIZE **-- Display program size --** **SIZE**

SIZE Filename

The SIZE command is used to display the memory requirements of a PROMAL compiled program without actually loading it into memory. The argument is the name of a legal PROMAL command file. The EXECUTIVE will read the header from the file and display the pertinent information in a form similar to the MAP command.

Example:

```
SIZE BILLIARDS
```

will display the load requirements of the BILLIARDS.C file without actually loading the program into memory. The SIZE command display will be similar to:

```
BILLIARDS   (PRO.) 9/21/84  VER 2
 CODE $06D7, GLOB VARS $0060, $10
```

The first line displays the command name, the kind of program (PROMAL), the compilation date, and the PROMAL version that created the object module (1 for pre-2.0, 2 for version 2.0 and above). The second line indicates the size of the executable program code in hexadecimal bytes, and the amount of memory required for arrays and simple variables, in bytes. No load address is shown, because the EXECUTIVE has not loaded the program.

TYPE -- **Display file on screen** -- **TYPE**

TYPE Filename

The TYPE command is used to display the contents of a **text** file or **source** program on the screen. It can also be used to display the contents of the Workspace. Output can be redirected to another PROMAL output device such as the printer. The argument is the name of the file or device to be typed.

Examples:

```
TYPE FIND.S
```

types the file FIND.S on the screen.

```
TYPE MYJOB.J >P
```

types the file MYJOB.J on the printer.

```
TYPE L
```

types the standard library (L device) on the screen.

```
TYPE K >P
```

turns your computer into an electronic correcting typewriter. All lines typed on the keyboard will be output to the printer when RETURN is typed. To terminate the command type CTRL-Z, which indicates "end of file" from the keyboard.

Notes:

1. If you attempt to type an compiled program or other non-text file, it will display garbage on the screen. This can happen if you forget to specify the ".S" extension on the filename to be typed.

2. On the **Commodore 64**, you may slow a TYPE display down by holding down CTRL. You may pause the display temporarily by holding down STOP. You can abort the command by pressing CTRL/STOP.

3. On the **Apple II**, you may temporarily halt the display with CTRL-S, and continue the display with any key. CTRL-C will abort the command.

UNLOAD -- Unload program from memory -- UNLOAD

UNLOAD [Commandname]

The UNLOAD command is used to remove a command (PROMAL compiled program) from memory, making space available for other programs. This will also make more room available for the EDIT buffer and the COMPILER's tables, which use the available space for temporary storage. The optional argument is the name of the command to be UNLOADED. If the command is not found, no error is indicated. If the command name is found, it will be removed from the EXECUTIVE's internal table of loaded programs, **along with any programs which occupy higher addresses.** If the **Commandname** argument is not specified, all commands are unloaded.

Example:

```
UNLOAD BILLIARDS
```

removes the BILLIARDS program from the EXECUTIVE's command list, freeing up memory previously allocated to it for other programs. The MAP command can be used to display the results.

Notes:

1. When a program is unloaded, the memory it occupied is not cleared or altered in any way. The space is merely deallocated, making it available for other programs as they are loaded.
2. If two loaded programs have the same name, only the last one loaded will be unloaded. This situation arises when different object file names were loaded which had the same PROGRAM name when compiled.

UNLOCK -- Unlock a file -- UNLOCK

AVAILABLE ON APPLE II ONLY

UNLOCK Filename [...]

The UNLOCK command removes the write-protect status from a file previously LOCKed. Once UNLOCKed, the file can be deleted, renamed, or overwritten. File names without an extension may be unlocked by enclosing the file name in quotes.

Example:

```
UNLOCK MYSOURCE.S
```

WS **-- Change or clear Workspace --** **WS**

WS Size ; This form is available on **Commodore 64** only
or
WS CLEAR ; This form is available on both computers.

Description:

The WS command is used to change the size of the Workspace (the W device in-memory file), or to discard its contents. The first form (for Commodore only) has an argument which specifies the desired Workspace size in hex bytes. If the value specified is less than $20, it is assumed to be in hexadecimal K-bytes instead. If the size requested is larger than the remaining available space, NOT ENOUGH MEMORY will be displayed. If the ACTIVE (in use) Workspace is larger than the requested Workspace, no action takes place and the message "ACTIVE W IS LARGER. (WS CLEAR WILL EMPTY W)" will be displayed.

The second form is used to clear the Workspace. The memory content of the Workspace is not actually altered, but the internal pointers used to manipulate the Workspace are set so that the active Workspace is 0.

Examples:

```
WS E
```

sets the Commodore Workspace size to 14K bytes ($E times $0400).

```
WS 1800
```

sets the Commodore Workspace size to $1800 bytes (6K bytes).

```
WS CLEAR
```

clears the active Workspace to 0 without affecting the size of the Workspace.

Notes for **Commodore 64**:

1. The default size of the Workspace is 2K bytes (2048 decimal). The location of the Workspace varies as programs are loaded and unloaded. The EXECUTIVE maintains the contents of the Workspace as it is moved around in memory. The COMPILER and EDITOR will use the Workspace for buffer space if it is completely empty. Therefore you can EDIT or COMPILE larger programs by issuing a WS CLEAR command first (See the EDITOR section of this manual for a more complete description of the EDITOR's treatment of the Workspace).

Notes for **Apple II**:

1. The workspace is maintained in auxiliary memory (the "other" 64K of banked memory). The Apple has a fixed size workspace of about 18K bytes.

THE PROMAL EDITOR

The PROMAL EDITOR is a full-screen text editor for preparing and changing text files. It incorporates many of the features found in the finest word processors, but is designed specifically for the generation of PROMAL source programs. Some of the important features of the EDITOR are:

* Cursor-driven, full-screen operation.
* Displayed function key legends and on-line HELP screen.
* Automatic vertical scrolling.
* Automatic horizontal line scrolling
* Insert or type-over mode.
* Global search and search-and-replace with "veto".
* Block copy, move, delete, save, and recall.
* Semi-Automatic indentation support for PROMAL programs.
* Fast operation.

The EDITOR is loaded into memory automatically when PROMAL is booted up, and is normally a permanent part of the PROMAL system in memory. This makes it very convenient to use, since the EDITOR is always instantly available.

ENTERING THE EDITOR

From the EXECUTIVE, you may enter the EDITOR with the EDIT command, of the form:

EDIT [Filename]

where **Filename** is an optional argument specifying the name of the file to edit. If the **Filename** is omitted, the EDITOR will use the file in the Workspace, if any. Otherwise, a new file will be assumed. If a **Filename** is specified and does not exist, the EDITOR will start a new file by that name. The EDITOR also assumes a default file extension of ".S", so it is not necessary to specify the extension when entering the file name.

The EDITOR begins by "signing on". If a new file is being created, this display will not be visible long enough to read because the screen will be immediately cleared and the cursor moved to the upper left-hand corner of the screen for you to start your new text file or program. If you are editing an existing file, the message will be visible until the file has been loaded into memory.

THE EDITOR DISPLAY FORMAT

For a new file, the EDITOR will show an initial display as shown in Figure 1a (For Commodore 64) or 1b (for Apple II) below.

FIGURE 1a

COMMODORE EDITOR SCREEN DISPLAY FOR A NEW FILE

```
----------------------------------------
 LINE=      1
F1=DEL LN F3=MARK    F5=FIND    F7=HELP
F2=INS LN F4=RECALL F6=CHANGE F8=QUIT
```

FIGURE 1b

APPLE II EDITOR SCREEN DISPLAY FOR A NEW FILE

```
-------------------------------------------------------------------------------
 LINE =    1
1=DEL LN  2=INS LN  3=MARK   4=RECALL   5=FIND   6=CHANGE   7=HELP   8=QUIT
```

The first 20 lines of the screen are initially blank for a newly created file, or contain the first 20 lines of an existing file. This area is called the **text area** and is where most editing operations take place. A full-width dashed line separates the text area from the bottom 4 lines of the screen, called the **status area.** This area displays the meanings of the current function keys and displays status information such as the current line number in the file. The line number display will change automatically as you move about in the file or insert, delete or move lines.

To create text in a new file, just type in the normal fashion. The RETURN key will advance the cursor to the next line. If you reach the end of the text area, the text area will automatically scroll up by one line to make room for additional text. The lines that scroll off the top of the screen are not lost but are merely no longer visible. If you move the cursor up beyond the top line, the text area will scroll down 1 line, until you stop moving the cursor up or reach the beginning of the file.

If the cursor stops advancing as you type and characters overprint one on top of the other, it means that either (1) you have reached the maximum line size so no more characters can be entered on this line, or (2) the edit buffer is completely filled.

In the EDITOR, you have all of the editing keys available to you that were in the EXECUTIVE, plus some additional ones. These keys are summarized in **Table 5** below.

The best way to learn how to use the EDITOR is to EDIT an existing file and experiment with the various keys. You will not alter the existing file on disk unless you specifically elect to overwrite the existing file when you exit from the EDITOR, so you won't do any harm.

EDITOR FUNCTION KEYS

The function keys operate somewhat differently in the EDITOR than in the EXECUTIVE. The meaning of each of the function keys is always shown in the status area of the screen. Under some circumstances, these function key "legends" will change during command processing, to allow more than a total of 8 possible actions. Some of the function keys perform an operation immediately, and some require additional input to be typed followed by RETURN. The action of each of the function keys is described in the following sections.

For the **Apple II,** function keys are activate by holding down either Apple key and pressing the desired number key.

THE HELP DISPLAY

Pressing the F7 (HELP) function key will temporarily erase the screen and display a screen showing the meanings of the control keys. Pressing RETURN will restore the normal display.

TABLE 5

PROMAL EDITING KEYS

Commodore Key	Apple Key	Description
RETURN	**RETURN**	End of line. Advances to the start of the next line. May be typed at any position in the line.
DEL	**DELETE**	Replace the character left of the cursor with a blank and backup the cursor one position . Will "pull back" characters to the right only if in "insert mode".
INST	**CTRL E**	Enable "insert mode". Any characters subsequently typed will be inserted before the character the cursor is on, pushing any existing text to the right. Exit insert mode by pressing RETURN or any cursor key.
CTRL ←	**CTRL D**	Delete character with pullback. Deletes the character under the cursor and pulls any remaining text to the left to fill in the gap. On the **Commodore 64,** this key is located above the CTRL key.
==>	-->	Cursor right. Moves the cursor to the right, without altering the character under the cursor. Stops at the end of the line. Repeats automatically after a brief pause if held down. On the **Commodore 64,** advancing beyond column 40 will cause the line to temporarily scroll left, allowing editing beyond column 40. On the Apple, line scrolling begins at column 80.
<==	<--	Cursor left. Moves the cursor to the left, without altering the character under the cursor. Stops at column 1 of the line. Repeats automatically after a brief pause if held down.
CRSR up	**up arrow**	Cursor up. Moves the cursor up by one line. If already at the top of the screen, scrolls the screen down to back up by one line, until the beginning of the file is reached.
CRSR down	**down arrow**	Cursor down. Moves the cursor down by one line. If already at the bottom of the text area, scrolls the screen up to advance by one line in the file, until end of file is reached. Will not advance beyond end-of-file (use RETURN to add new blank lines at end-of-file).

Note: N/A means not available. CTRL X means hold down the CTRL key and press X.

TABLE 5 (continued)

Commodore Key	Apple Key	Description
HOME	**CTRL Y**	Position the cursor to the upper left hand corner.
CTRL X	**CTRL X**	Clear the entire line. Erases all characters typed on the line and repositions the cursor to the first column of the same line.
CTRL K	**CTRL **	Clear to end of line. Erases all characters from the cursor to the end of line.
CTRL Y	**CTRL L**	Jump to last character on line. Moves the cursor to the character after the last character on the line, without affecting the line content.
CTRL [	**CTRL F**	Jump to first character on line. Moves the cursor to the first character position, without affecting the line content.
CTRL A	**CTRL A**	Toggle Alpha-Lock mode. Switches in or out of ALPHALOCK mode. If the ALPHALOCK indicator appears in the status area, then all subsequent alphabetic characters to be entered and displayed as upper case when typed. Pressing CTRL A again returns to normal upper and lower case alpha mode.
CTRL I	**TAB** or **CTRL I**	Tab and indent. Moves the current level of indentation in by two columns, forming a new temporary margin. Generally used after a conditional statement in a PROMAL program.
CTRL U	**CTRL Q**	Un-indent. Moves the current left margin two columns left, so that subsequent text will be entered with one less level of indentation. Generally used to end an indented block following a conditional statement in a PROMAL program.
CTRL N	**CTRL N**	Next page. Erases the text area and displays the next 20 lines from the file being edited.
CTRL P	**CTRL P**	Previous page. Erases the text area and displays the prior 20 lines from the file being edited.
CTRL J	**CTRL ^**	Adjust right. Moves the line the cursor is on to the right by one level of indentation (2 columns) and advances the cursor to the next line. Usually used to indent an existing block of statements in a PROMAL program when adding a conditional statement in front of the block.

Note: N/A means not available. CTRL X means hold down CTRL key and press X.

TABLE 5 (continued)

Commodore Key	Apple Key	Description
CTRL O	CTRL O	Adjust left. Moves the line the cursor is on to the left by one level of indentation and advances to the next line. Usually used to remove a level of indentation from a PROMAL program.
CTRL W	CTRL W	Set window. Erases and redisplays the text window starting with the cursor's current column. Usually used to examine or edit the right-hand portion of a group of lines which are wider than the screen. Does not effect the line content. The characters in leftmost column will be highlighted to emphasize that text exists off the left side of the display.
CTRL V	CTRL V	Normalize window. Restores the normal text display position so that lines are displayed beginning with column 1. Used to remove effect of a prior CTRL W.
CTRL B	CTRL B	Backtrack. Erases any existing command line and enters the last EDITOR command entered. More can be typed after the recalled command, or it can be edited further. Pressing CTRL B again will recall the next-to-most recent line entered. This can be repeated up to the limit of the backtrack buffer of 256 characters; then the display will "wrap around" to repeat the most recent command again.

Note: N/A means not available. CTRL X means hold down the CTRL key and press X.

INSERTING AND DELETING LINES

Pressing the F1 (DEL LN) function key will cause the line the cursor is on to be deleted and the lines below it will move up to fill in the gap. If you delete a line accidentally, you can recover it by pressing the F4 (RECALL) function key followed by the RETURN key.

Pressing the F2 (INS LN) function key will open up a new, blank line above the line the cursor is currently on, and move to the start of the new blank line. Existing lines of text are pushed down to make room for the new line.

SEARCHING WITH THE FIND KEY

The F5 (FIND) function key can be used to jump to a specific line number in the file or to locate a particular character string. If you press F5, the word FIND will appear in the status area, followed by a blinking cursor. If you type in a number followed by RETURN, the text area will be immediately redisplayed starting with that line number. For example:

```
FIND 67
```

will immediately display lines 67 through 86 of the file. You can also jump to the end of the file by merely typing a number greater than the last line of the file, for example FIND 9999. The largest line number that can be entered is 32767. You may jump to the beginning of file using a FIND 1 command.

You may also enter a displacement instead of an absolute line number. For example,

```
FIND +50
```

will redisplay the text area beginning 50 lines from where you are now, and:

```
FIND -200
```

will back up 200 lines from where you are now.

You may search for a certain string with the F5 function key by specifying the desired string in quotes. For example:

```
FIND 'INFILE'
```

searches for the string "INFILE". Either single (') or double (") quotes may be used to enclose the string, but must match on both ends. Searches are **always made in the forward direction, starting from the present cursor position** (not necessarily at the top of the screen). If the string is found, the text area will be redisplayed starting with the line containing the string, with the cursor on the first character of the string. If the string is not found, a "NOT FOUND" message will be displayed in the status area and the text area will not be changed. Press RETURN in this case to remove the message.

You may also combine searches on a single command. For example to jump to the beginning of file and then search for 'WHILE GETC', you could press the F5 function key and complete the command as:

```
FIND 1 "WHILE GETC"
```

If "WHILE GETC" is not found, then the text area will redisplay starting at the first line after the NOT FOUND message. In a compound search such as this, the screen will display the last "successful" part of the search (that is, the FIND 1 part in this case). **A particularly useful compound search is:**

```
FIND +1 'XXXXX'
```

where XXXXX is whatever text you are looking for. If this finds an occurrence of XXXXX, but not the one you wanted, you can repeat the search starting with the next line by just pressing CTRL B and RETURN. You can also use a compound search to locate a selected word in a particular subroutine. For example:

```
FIND 1 'PROC SUB1' 'ARG'
```

will find the first occurrence of 'ARG' in procedure SUB1, but will ignore all prior occurrences in the file, because the EDITOR will first jump to line 1, then search for 'PROC SUB1', and then search forward for 'ARG'.

SEARCH AND REPLACE

Function key F6 (CHANGE) is similar to the FIND function key, but allows you to change a string to a different string automatically. Like the FIND command, you will need to complete the command after pressing the F6 key. For example:

```
CHANGE 'SPRITEXY  'SPXY'
```

will search forward from the present cursor location for the string "SPRITEXY", and when it is found will highlight the matching string in the text window and display the prompt:

```
CHANGE THIS STRING (Y/N/C=CANCEL)?_
```

Replying with the Y key will cause the string to be replaced and the command completed. Replying with the N key will cause the EDITOR to search for the next occurrence of the string, and repeat the prompt when it is found. Pressing the C key will abort the command with the message "0 CHANGES MADE".

More often, you will want to use the CHANGE command to replace several or all the occurrences of a certain string. This can be done by replacing a repeat constant in front of the string to be searched for. For example:

```
CHANGE 999 "SPRITEXY" "SPXY"
```

will tell the EDITOR to replace up to 999 occurrences of the string "SPRITEXY" with "SPXY", again searching forward from the present cursor location. The EDITOR will pause at each occurrence, highlight it, and prompt:

```
CHANGE THIS STRING (Y/N/C=CANCEL)?_
```

This gives you a chance to "veto" any occurrences which you don't want to change. This is very important, since the EDITOR will often "surprise" you with occurrences you didn't think of. For example, "TEN" may appear in "OFTEN", which you really didn't want to change. After the last occurrence is found, the EDITOR will tell you how many changes were made.

"CUT AND PASTE" OPERATIONS

The PROMAL EDITOR supports a variety of block operations, often known as "cut and paste" operations. A block of text is one or more complete lines of text to be operated on as a group. Blocks can be moved, copied, deleted, saved to a file or inserted from a file. To designate a block, place the cursor in any column of the first line of interest and press the F3 (MARK) key. The line will be highlighted, and the function key legends will change to:

```
F1=DELETE  F3=MARK   F5=FIND  F7=COPY
F2=        F4=WRITE  F6=MOVE  F8=CANCEL
```

which indicate your new choices (shown on one line on the Apple II). Move the cursor to the last line of the text block and press F3 (MARK) again. For large blocks you may need to scroll the screen or use a FIND command to locate the last line desired. All intervening lines will be highlighted when you press the F3 key again. Any number of lines can be marked. You have now designated the block to operate on. The F8 (CANCEL) key can be used at any time to cancel the command. You may also mark the block with the last line first and then the first line; it doesn't matter. If you wish, you can also just press F3 repeatedly until you have the desired number of lines marked. Each depression of F3 will mark one more line.

If you want to move the block or copy it elsewhere in the file, position the cursor to the desired destination and press F7 (COPY) or F6 (MOVE). The marked block will be inserted above the line the cursor is on. The high-lighting will be removed, completing the command. Copying a block requires as much free space as copying. Therefore when moving large blocks, you may wish to use WRITE block, DELETE block, RECALL block instead (described below). If you want to delete the marked block, press F1 (DELETE) instead. **CAUTION: THERE IS NO WAY TO "UN-DELETE" A DELETED BLOCK.**

SAVING A BLOCK TO A FILE

If you want to save a marked block to a file, press the F4 (WRITE) key. The word "WRITE" will appear in the status area, followed by the cursor. Complete the command by typing the name of a legal PROMAL file to receive the copy of the marked block. The file will be created and written on disk, and the highlighting removed, completing the command. For example:

```
WRITE TEMP.S
```

will copy the marked block into the file TEMP.S on disk. (Note: if you omit the extension, the EDITOR will supply a ".S" extension by default.) You may also write to devices. For example:

```
WRITE P
```

will type the marked block on the printer. You may also write a block to the Workspace, under some circumstances (see the discussion of the workspace and the edit buffer at the end of this section for more information). You cannot append to the Workspace, however. Do not try to write to the S device.

To insert a file on disk into the file being edited, move the cursor to the desired location and press F4 (RECALL). Complete the command by typing in the name of the desired file. The file will be inserted above the present cursor position. To insert a file after the very last line of text, press RETURN to put the cursor on a new line below the last line; then do the insert.

When you start writing your own PROMAL programs, you will find the BLOCK-WRITE and RECALL commands very useful for copying pieces of an existing program into a new program, so you don't have to type all those lines over.

EXITING FROM THE EDITOR

Function key F8 (QUIT) will display the buffer status and exit menu,

for example:

```
     PROMAL EDITOR VERSION 2.1
   COPYRIGHT (C) 1986 SMA, INC.

FILE NAME = W (AUTO-UPDATE ON QUIT)
BUFFER SIZE = 10685
FILE SIZE = 9596


     R = REPLACE ORIGINAL FILE

     N = WRITE TO NEW FILE

     W = WRITE TO WORKSPACE

     C = CONTINUE EDITING

     Q = QUIT EDITOR

SELECTION?

_
```

The buffer size and file size are shown in decimal bytes. The buffer size represents the maximum file size that the EDITOR can work on (it can be increased by unloading programs or, on the Commodore 64, by clearing the Workspace).

Press the key corresponding to the desired option and press RETURN. R will replace the original file you specified on entry to the EDITOR, if any (**it may be a new file you specified on entry to the EDITOR, too**). If you select N, you will be prompted to enter the name of the new file. The W option writes to the Workspace (W device). It is possible that the Workspace may not be large enough to hold the entire file edited. In this case, you will be asked if you wish to write to the Workspace anyway, in which case the file will be truncated. Otherwise you can write the whole file to the disk to save it. On the Commodore 64, when "auto-updating" the Workspace, the Workspace may be increased in size automatically if needed to hold the file.

For convenience, the Workspace is **automatically updated** when you Quit the EDITOR if you entered the EDITOR **without** specifying any file or device name. This is so that you will have your corrections saved in the workspace without having to remember to select the W option explicitly. (NOTE: You can edit the workspace without having it automatically updated on exiting from the editor, by starting the EDITOR with the command "EDIT W" instead of just "EDIT").

After you have made one selection, such as writing the file to disk, you can make another selection. The C option will let you continue editing the file. It is a good idea when you are doing a lot of editing to periodically write the file out to disk so all is not lost in the event of a power failure or similar problem. When you are ready to return to the PROMAL EXECUTIVE, enter Q and RETURN. This will clear the screen and return to the EXECUTIVE. NOTE: Never abort the EDITor with CTRL-STOP (Commodore) or CTRL-RESET (Apple).

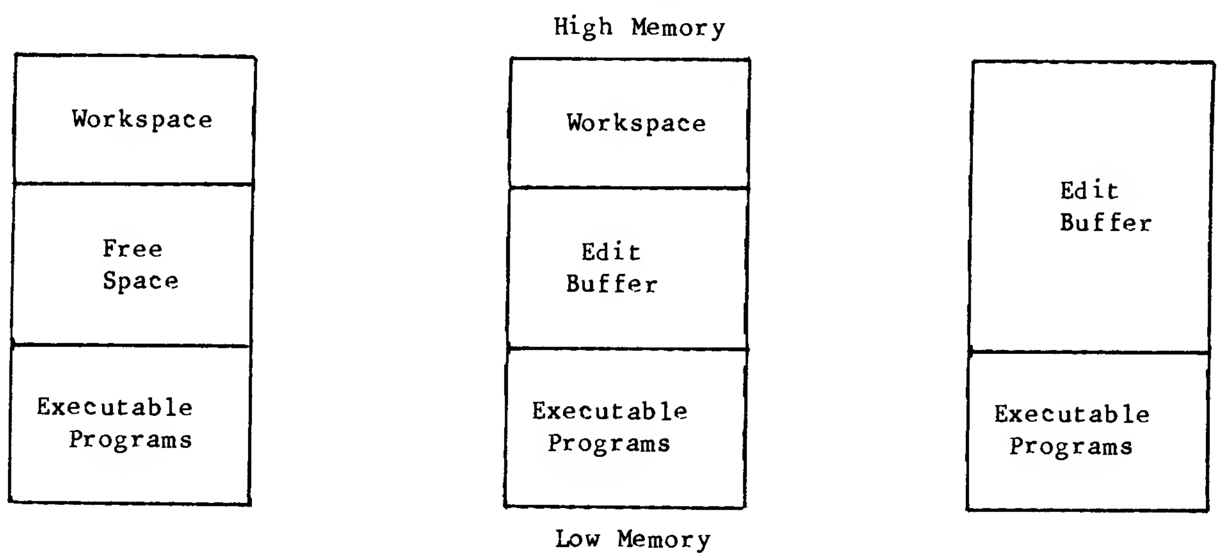
As a final note, the PROMAL EDITOR is itself written in PROMAL. We think that you will find this fact significant if you have ever considered writing your own editor, word processor, or similar program in a high-level language. Obviously, PROMAL can do the job!

TECHNICAL NOTES ON THE EDITOR FOR THE COMMODORE 64

The remainder of the section on the EDITOR applies only to the **Commodore 64.**

RELATIONSHIP OF THE WORKSPACE TO THE EDIT BUFFER

The EDITOR uses all unallocated memory as a buffer for editing your text. The size of unallocated memory can be displayed by the EXECUTIVE MAP command. Depending on how you enter the EDITOR and whether the Workspace is empty or not, the Workspace (W) may or may not be "absorbed" into the edit buffer:



Before EDITing

EDITing with Workspace preserved

EDITing with Workspace absorbed

The Workspace will be "absorbed" into the Edit buffer (to maximize your available editing buffer size) if:

1. The Workspace is empty, or,

2. The Workspace is not empty and NO file name was specified on the EDIT command (in this case the Workspace is absorbed non-destructively).

The Workspace will NOT be absorbed into the Edit buffer if:

1. The workspace is NOT empty and a file name or W was specified on the EDIT command.

If the Workspace is absorbed, you cannot WRITE or RECALL from the Workspace during the EDIT session. If the session was started with an EDIT command without a file name or W specified, the Workspace will be automatically updated when you quit the EDITOR. If the Workspace is not absorbed, you may use it like a file during the EDIT session (for SAVEs and RECALLS).

In any event, the Workspace will be "un-absorbed" when you exit from the EDITOR back to the EXECUTIVE. The EXECUTIVE WS command can be used to clear the Workspace or alter its size before entering the EDITOR.

EDITOR COMMODORE 64 CHARACTER SETS

The EDITOR creates and updates files of standard ASCII characters. Unfortunately, the Commodore character-generator ROM's do not conform completely to the ASCII standard. As you may know, there are two sets of characters which can be displayed on the screen, upper case plus graphics and upper plus lower case. The PROMAL EXECUTIVE selects upper and lower case mode, and this is the mode that should normally be used with PROMAL. The ALPHA LOCK (CTRL A) feature can be used to automatically generate upper case letters in this mode, if you so choose.

There may be times, however, when you might want to edit in the upper case and graphics mode. You can switch modes by pressing SHIFT C= (shifted Commodore key). When the EDITOR starts up, it senses the currently selected mode and edits accordingly. Also, when the EDIT HELP menu is displayed, you may switch modes and then resume editing with the C selection. Selecting the upper case and graphics mode will permit you to enter the various non-ASCII characters supported by Commodore. For installing special characters into strings in a PROMAL program (such as to change colors or reverse video), we recommend that you use the special hex literal form described in the PROMAL LANGUAGE MANUAL instead of using graphics keys (for example, "£12 Hello! £92" will generate a reverse video string, " Hello! "). When you exit back to the PROMAL EXECUTIVE, whatever character set you entered the EDITOR with will be restored.

THE PROMAL COMPILER

The heart of the PROMAL system is the PROMAL compiler. This program takes as input a file containing your source PROMAL program (which you normally generate with the editor). It produces as output a small, very-fast executing command file (object program). This object program can be executed from the PROMAL EXECUTIVE by merely typing its name. Optionally, the COMPILER can also produce a listing file. The listing shows your original source program, with line numbers and the addresses assigned to variables and statements.

Unlike BASIC, you cannot execute a program in the form you type it in. First it **must** be compiled. This is a great benefit, since the compiled program will only be a small fraction of the original size, and will execute many times faster than it would be possible to execute the original text version of the program with an interpreter, as is done in BASIC. The compiler does not alter your source file; it is left intact so you can examine it or edit it further. Anytime you make a change to the source program, you will need to re-compile it before the changes will take effect in the object program.

Unlike the EDITOR and the EXECUTIVE, the COMPILER is not automatically loaded into memory when PROMAL boots up. This is because the COMPILER is a large program (about 15K bytes), and it is only needed when you want to compile a new program (or revised old program) into executable form.

STARTING THE COMPILER

The COMPILER can be run from the EXECUTIVE with a command of the form:

COMPILE [Sourcefile][**O**=Objectfile]][**L**[=Listfile]][**V**=c][**B**[=Solf]]

The simplest form of this command is just:

COMPILE

which will compile a source program from the Workspace and produce an executable object program on disk, ready to run by typing its name. The object program written will have the same file name as the name on the PROGRAM line of the source program, with a ".C" extension.

If you compile a program which already has an existing object file on the disk, the compiler will prompt:

REPLACE EXISTING Filename.C (Y/N)? _

Press Y and RETURN to replace the old file; any other reply will terminate the compilation. After a successful compilation, the EXECUTIVE will automatically unload any existing copy of the compiled program from memory. This ensures that you will always execute the new version of a freshly compiled program, not an old version which might still be in memory.

For the **Commodore 64:** The demo compiler can be used to compile small programs directly from the Workspace. **When using the full compiler (which is larger than the demo compiler), you will want to keep the Workspace clear and compile directly from disk.**

For the **Apple II:** The compiler is always loaded from disk when needed, and unloads when it is finished, so you will have more room available for your programs. It is a good idea to give an UNLOAD command before compiling.

Most frequently you will compile directly from a source program on disk and produce an object file on disk. For example:

```
COMPILE MYPROG
```

will tell the compiler to read the file MYPROG.S (the compiler supplies the ".S" default extension) and produces an object file called MYPROG.C. The COMPILER gives the object file on disk the same name as the source file, but with a ".C" extension.

You can use the "O" (the letter "oh") option to output the object code to a file other than the same name as the source file. For example:

```
COMPILE MYPROG O=NEWVERSION
```

will read MYPROG.S and generate NEWVERSION.C. (Note: we strongly recommend that you **use the same file name for your object file as is declared on the PROGRAM header line of the source program.** Otherwise, the EXECUTIVE will load the FILE you specify from disk each time you type its name even if the same program is already in memory, because it does not match the command name. The name on the PROGRAM statement is the name which appears in the MAP display.)

If you add the "L" option to your command, a listing will be produced. The "L" option by itself will cause the listing to be written to a file with the same name but with a ".L" extension. For example:

```
COMPILE MYPROG L
```

will read MYPROG.S, produce an object file MYPROG.C, and listing file MYPROG.L. If the listing already exists it will be replaced. You can also send the listing directly to the printer by specifying **L=P.**

The V option is used to specify a version of your program for conditional compilation, described in Chapter 5 of the PROMAL LANGUAGE MANUAL.

The final option for the COMPILE command is the "B" option, which is used to compile big programs from disk. This option will not normally be needed unless you are compiling fairly large programs. The B option is not available on the Demo compiler. It is easy to tell when you need to specify the B option. The statistics displayed at the end of a compilation tell you how much of the COMPILER's available table space was used. If any of the percentages shown start approaching about 90 percent, you probably should be using the B option. When compiling large programs, <u>you should be sure to remember to UNLOAD</u> other programs before compiling, because the COMPILER uses all available memory for its tables. For example:

```
UNLOAD
COMPILE MYPROG B
```

will read MYPROG.S and produce an object code file MYPROG.C, allowing <u>very</u> large programs to be compiled (typically several thousand lines).

The B option changes the way the compiler uses memory, so it will have more memory for its tables. Normally the compiler stores the compiled object code in a buffer in memory until the whole program is compiled, and then writes it to disk. When the B option is selected, it writes the object code to disk "on the fly" instead. For the **COMMODORE 64** only, the "B" option will also delete the EDITOR from memory, making enough room available to compile very large programs. The EDITOR will be reloaded from disk when you need it.

For **very** large programs it is possible that one of the tables used by the compiler may overflow even with the B option. For example, if your program uses many hundreds of lines of strings in data statements, the string table may get full (as indicated by the statistics displayed at the end of compilation). In this case, you can tell the compiler to allocate more of available memory for the particular table which is overflowing, using this form of the B option:

```
COMPILE Myprog B=Solf
```

where **Solf** represents a four digit hexadecimal number, where the four individual hex digits must sum to exactly 10 hexadecimal. For example:

```
COMPILE MYPROG B=C112
```

Each digit represents what part of available memory is to be allocated for a particular table, in units of 1/16th of available memory, as follows:

S (first digit) = number of 16ths used for symbol (name) table
O (second digit) = number of 16ths used for object buffer
L (third digit) = number of 16ths used for literals (strings)
F (fourth digit) = number of 16ths used for forward references

Therefore the example above used 12/16 ($C=12 decimal) of available memory for the symbol table, 1/16th for the object code buffer, 1/16th for strings, and 2/16ths for forward references. The default for the B option is B=A132. The value of the second (O) digit should always be left at 1.

For the **COMMODORE 64**, you should never attempt to use the L option to produce a listing file on disk at the same time as using the B option. This is because the 1541 disk drive will only allow one file to be open for writing at a time while a file is open for reading, and the B and L options each require one output file. If you want to get a listing, use L=P or make a second compilation with O=N to suppress the object file.

DIALOGUE OF THE COMPILATION PROCESS

When the COMPILER starts, it will run continuously until your program is compiled or until an error is detected, without further action on your part. After it "signs on", it will immediately read in your your source program, and show a rapidly changing message:

```
READING LINE nnn
```

where nnn is the line number the compiler is reading. This number will change rapidly (several lines per second), especially if you are compiling from the Workspace. When the compiler encounters an INCLUDE LIBRARY statement in

your program, it will display another line saying:

```
INCLUDING LIBRARY
```

and will then generate another line of the form

```
READING LINE nnn
```

as it scans the LIBRARY. This takes about 5 seconds. It will then display:

```
RESUMING FILE 0:Filename.S
```

indicating that it has finished the INCLUDE file and is resuming reading the original file (the "0:" indicates the disk drive number for Commodore 64 versions; for Apple II this will be replaced with the path name). Also, each time the COMPILER reads a new Procedure or Function definition in your program, it displays a message showing the start of the subroutine.

If an error is detected in your program, the COMPILER will halt and display the offending line, followed by an error message. For example:

```
READING LINE 303
COMPILING FUNC BYTE READJOYSTICK
READING LINE 309
       IF FIREBUTTON OR UP OR DOWN
*********************

ERROR 27:
UNDEFINED

Want to EDIT W (Y/N)? _
```

This indicates that the variable FIREBUTTON has not been defined (declared). The row of asterisks indicates how much of the line the COMPILER had read before the error was detected.

The line "Want to EDIT W (Y/N)?" asks if you want to edit the file containing the error (in this case, the Workspace, W). If you reply with Y (and <RETURN>), the EDITOR will start and automatically position the cursor to the offending line, simplifying the correction process. Otherwise, you will be returned to the EXECUTIVE.

When looking at a compiler error message, note that the row of asterisks may not always indicate the **exact** location of the problem. The asterisks only indicate how far the compiler had read before it recognized an error. The error could have been caused by anything up to this point in the program. For example, a misspelled variable declaration near the beginning of the program will not cause an error until referenced with the "correct" spelling later in the program. **Appendix C** contains a listing of all Compiler error messages with an explanation and corrective action for each. The compiler error messages are contained in file COMPERRMSG.T. If this file is not present on your disk, you will only get an error number, without the text of the message.

Assuming your program compiled to completion without an error, you will see a signoff message with a summary. This indicates that your program compiled

successfully and is ready to execute. You are back in the EXECUTIVE. The summary shows the total number of lines read by the compiler, the number of bytes of object code generated, the number of bytes of memory your program will need for variables, and how much of the compiler's tables were used. Numbers with a $ prefix are hexadecimal, and those in parentheses are in decimal. Note that the actual object file will be somewhat larger than the number of bytes of object code generated, because the object file contains additional header and relocation tables.

DIFFERENCES BETWEEN VARIOUS PROMAL COMPILERS

The Demo compiler and standard compiler are both named COMPILE. The Demo compiler is found on the demo disk and the full compiler on the PROMAL system disk. You may prefer the Demo compiler for small program development.

The Demo compiler does not support the B option (described above), nor does it support IMPORTs, EXPORTs, or separate compilation (described in Chapter 8 of the PROMAL LANGUAGE MANUAL), and is limited to a maximum of 400 source lines (excluding comments but including the LIBRARY).

For the **Apple II,** if you use the EXECUTIVE command GET COMPILE, the standard compiler will not show up in the memory MAP and cannot be executed from memory, because there is not enough room in memory for the compiler and the EXECUTIVE at the same time. This is of no consequence, except that you need to keep a copy of the compiler on each of your disks you use for program development.

PROMAL CROSS REFERENCE MAP UTILITY

XREF generates an alphabetized list of all the identifiers (names) used in a program, with a sorted list of line numbers on which they appear. This list is very useful for locating where variables, procedures, or functions are used in a program. The XREF utility is normally used after a compilation with a listing generated. In this case, XREF will append the cross-reference map to the end of the listing file. The syntax of XREF is:

```
XREF [Sourcefile][V=c]
```

where Sourcefile is the name of the PROMAL sourcefile to be read. It may be good idea to UNLOAD any other programs before running XREF, so it will have plenty of room for the tables it uses internally. The optional second argument **V** specifies a version for conditional compilation. If you are using conditional compilation, you should specify the same V option that you use on the corresponding COMPILE. Omitting the V will cross reference all conditional blocks.

When XREF is run, it appends the cross reference map to the end of the listing file, if one exists. Otherwise, it generates a listing with line numbers (but does not generate any code or do any syntax checking like the compiler) and a cross reference map, on a file with the same name as the source file but with a **.X** extension. This is useful if you wish to obtain a cross reference map for a program which will not successfully compile yet.

P R O M A L

(PROgrammer´s Micro Application Language)

LANGUAGE MANUAL

A PROMAL LANGUAGE DESCRIPTION AND REFERENCE

For Apple II and Commodore 64 Computers

SYSTEMS MANAGEMENT ASSOCIATES, INC.
3325 Executive Drive
Raleigh, North Carolina 27609
(919) 878-3600

Rev. C - Sep. 1986

PROMAL LANGUAGE MANUAL

CHAPTER 1: INTRODUCTION

This manual will introduce you to the PROMAL programming language, which we think will find to be the most enjoyable and creative language available for your computer. This manual will guide you step by step through a description of the PROMAL language, with examples along the way. It assumes that you already have a working knowledge of BASIC (or some other high-level language) and elementary computer concepts such as bits, bytes and memory addresses. Comparisons are often given between PROMAL programs and the equivalent BASIC program, so that you may draw on your previous experience.

You should study the manual carefully, because PROMAL is significantly different from BASIC. As a BASIC programmer, you may find some aspects of PROMAL a little strange at first reading. But if you give it a fair trial, we´re sure you will soon want to do all of your programming in PROMAL.

We assume that you have already read the companion manual MEET PROMAL!, which provides a "hands-on" introduction to the PROMAL system as a whole. You will find the operational aspects of the PROMAL EXECUTIVE, EDITOR, COMPILER, and LIBRARY described in detail in the PROMAL USER´S MANUAL. This manual explains the heart of the PROMAL system, the PROMAL programming language, which you can use to create your own programs. PROMAL is especially well suited for:

* Text processing applications
* Scientific and Engineering applications
* Educational applications
* Interactive programming
* Small business programming
* Compilers, assemblers, editors or system software

Not only do PROMAL programs for these applications often run 20 to 100 times (or more) faster than BASIC, they are actually easier to program than BASIC! Programs that used to take weeks or months of assembly-language drudgery can now be quickly developed with PROMAL instead.

WHY USE PROMAL?

Why should you learn PROMAL when you already know BASIC? Why should you learn PROMAL instead of one of the older, structured languages such as PASCAL or C?

Perhaps the most important reason is that PROMAL is in many respects the **most structured** language available, because the PROMAL compiler reads indentation as part of the syntax of the language. As you will see, the fact that indentation **always** shows the true structure of your program will make your programs easier to write, and more importantly, **easier to maintain.**

Another important consideration is that PROMAL is the **only** compiled language available from a single vendor for the IBM PC, Apple II, and Commodore 64 - the three biggest-selling machines in history. If you plan to develop commercial software, or just think you might change computers some day, this will be important to you. And PROMAL gives you top performance on all machines.

CHAPTER 2: PROMAL PROGRAMMING LANGUAGE OVERVIEW

A PROMAL **source program** is a file of text composed of lines, normally created using the PROMAL EDITOR. Each line is called a **statement**. A program has a certain organization to it, which is similar to a recipe. A program starts by declaring its name, and then identifies what "ingredients" are used in the program. Ingredients are identified by the kind of data to be used, the name of the data, and the quantity required. The list of ingredients is called the **declarations** part of the program. After the declarations part of the program comes the actual instructions which tell how to manipulate the data. For clarity, the instructions are usually broken up into a number of **procedures**, each of which has a name suggestive of its function.

For example, consider the actual PROMAL program in the right column, below, and observe the similarities with the recipe at the left.

A Kitchen Recipe		A PROMAL Program
FRIED CHICKEN:	<-- Your recipe name -->	PROGRAM LONGESTLINE INCLUDE LIBRARY
2 lb. Chicken pcs. 1/4 lb. shortening	<-- Main ingredients --> and amounts needed	BYTE LINE [81] ;current line BYTE LONGEST ;longest length WORD IFILE ;input file
SEASONED FLOUR:	<-- Sub Procedure Name -->	PROC SIZELINE
1/2 cup flour 1 tsp. salt 1/4 tsp. paprika	<-- Ingredients for --> sub-procedure	BYTE LENGTH ;cur line length
Mix all ingredients for seasoned flour.	<-- Instructions for --> sub-procedure	BEGIN ; Procedure LENGTH=LENSTR(LINE) IF LENGTH > LONGEST LONGEST = LENGTH END
Heat oven to 450. Melt shortening. Coat chicken with seasoned flour.	<-- Main Process --> setup	BEGIN ; Main Program IFILE=OPEN("TESTFILE.T") LONGEST=0
Cook about 45 min. until golden brown.	<-- Loop waiting for --> a condition	WHILE GETLF(IFILE,LINE) SIZELINE ;test if biggest
Serve with gravy.	<-- How to serve up --> the results	OUTPUT "Longest = #I",LONGEST END

The program above reads a file and prints the length of the longest line in the file. It is useful to get the feel for what a complete (although very simple) PROMAL program looks like before delving into the details.

If you have programmed in BASIC, probably the first thing you will notice about the program above is that there are no line numbers. Line numbers are not used and not needed in PROMAL programs. You will soon discover that this makes PROMAL programs much easier to write and understand. **PROMAL statements normally start in column 1.** Let's look briefly at the statements that compose the program, just to get the general idea of what they do.

```
PROGRAM LONGESTLINE
```

This line starts the program. The name LONGESTLINE is the command you will eventually type from the EXECUTIVE when you want to run this program.

```
INCLUDE LIBRARY
```

This line tells the PROMAL COMPILER to include the definitions of all the built-in library routines (which are needed for input-output, etc.). You will normally have this statement near the start of every program.

```
BYTE LINE [81]       ;current line
```

This line declares that you will be using a variable called LINE which is an array of 81 BYTEs. One byte can store one character, so this array can hold an 80 character line plus a line terminator. The ";" indicates the start of a comment. The rest of a line after a ";" is ignored by the compiler.

```
BYTE LONGEST         ;longest length
```

This line declares a simple (non-array) variable called LONGEST. It is used to hold the number of characters in the longest line. In PROMAL, unlike BASIC, **all variables must be declared before they are used** (not just arrays).

```
WORD IFILE           ;input file
```

This line declares a variable of type WORD. Later you will learn that a WORD is usually used to hold an address. In this case, the address will be a **"file handle"** for the file of text to be read. You can think of a file handle as just a number identifying a particular file.

```
PROC SIZELINE
```

This line begins the definition of a PROMAL procedure, which is similar to a BASIC subroutine. It has been given the name SIZELINE by the programmer.

```
BYTE LENGTH          ;cur line length
```

This is a variable of type BYTE which is only used within the procedure. This will be explained further later on.

```
BEGIN
```

This line signals the beginning of actual executable instructions within the procedure.

```
LENGTH=LENSTR(LINE)
```

This is an **assignment statement** which uses the built in function LENSTR to determine the number of characters currently in the array LINE, and install that number into LENGTH.

```
IF LENGTH > LONGEST
  LONGEST = LENGTH
```

The IF statement tests if the current length is larger than the largest line length so far, and if so, updates the value of LONGEST to LENGTH. Otherwise, the second statement is simply skipped over.

```
END
```

The END statement indicates the end of the procedure (like a BASIC RETURN statement).

```
BEGIN
```

Since there are no more PROCEDURES, the BEGIN signals the beginning of the main program. In PROMAL, **the main program always comes last.** This may seem a little strange at first, but follows from the general rule that everything, including all subroutines, must be defined before being used.

```
IFILE=OPEN("TESTFILE.T")
```

This is actually the first statement which would be executed in the program. It tells the computer to "OPEN" the file called "TESTFILE.T" for reading, and installs the file handle into IFILE. Any subsequent input references to IFILE will read from "TESTFILE.T".

```
LONGEST=0
```

This statement works just like its BASIC equivalent, and initializes the value of LONGEST to 0.

```
WHILE GETLF(IFILE, LINE)
  SIZELINE   ;test if biggest
```

These two statements comprise a **loop.** The WHILE statement attempts to read one line from the file into the LINE array. If successful, the SIZELINE subroutine is called, and the WHILE statement is repeated again. This process is repeated until end-of-file is reached, in which case the GETLF function is unsuccessful, and control passes through without executing SIZELINE again. In PROMAL, **subroutines are called by merely typing their names**; no GOSUB is needed.

```
OUTPUT "Longest = #I", LONGEST
```

This statement is similar to a BASIC PRINT statement. It would show an answer on the screen, for instance:

```
Longest = 67
```

assuming the longest line was 67 characters. The "#I" in the OUTPUT statement is a code which tells the computer **how** to format the answer; in this case, telling it to print it as an integer number.

END

This line terminates the program.

It is not important to understand the details of the program at this point, but just to get the general idea of what a program looks like. The following sections will explain the rules for writing a program in detail.

CHAPTER 3: ELEMENTS OF THE PROMAL LANGUAGE

In the last chapter we got a quick "top down" view of how a simple complete PROMAL program looks. In this chapter, we will take a "bottom up" look at some of the elements of the PROMAL language in greater detail. Then we will learn how to combine these elements into statements and programs.

VOCABULARY

The following **reserved** words have special meaning in PROMAL programs, and form the basic vocabulary of the language:

AND	CON	EXT	INT	OWN	TO
ARG	CHOOSE	FALSE	LIST	PROC	TRUE
ASM	DATA	FOR	NEXT	PROGRAM	UNTIL
AT	END	FUNC	NOT	REAL	WHILE
BYTE	ELSE	IF	NOTHING	REFUGE	WORD
BEGIN	ESCAPE	IMPORT	OR	REPEAT	XOR
BREAK	EXPORT	INCLUDE	OVERLAY	RETURN	

The reserved words may be spelled with either upper or lower case letters, or a mix of both. Therefore BEGIN, begin, Begin, and BegIn are equivalent. These reserved words are also sometimes called **keywords**. In PROMAL (unlike BASIC) you **must** separate keywords from each other or from other names with blanks or other punctuation. This helps make programs readable and does not impose any speed or memory size penalty on the program. As a practical matter you may also wish to consider the standard library routine names listed at the start of the LIBRARY MANUAL as reserved words, although this is not strictly true because you do not have to use the LIBRARY. You may even change the names in the LIBRARY, although this is definitely not recommended (for reasons of consistency with other programmers).

NAMES

Names are used to identify constants, variables, data, functions, procedures and programs in PROMAL. You may choose names (also called identifiers) as you wish, following these rules:

1. A name may be from one to 31 characters in length.
2. The first character must be alphabetic.
3. The remaining characters must be alphabetic, numeric, or the underline character "_" (left-pointing arrow on the Commodore 64, which has no underline key).
4. Either upper or lower case alphabetic characters may be used. Both are considered equivalent. The PROMAL compiler treats all alphabetic characters as upper case in identifiers. Therefore XYZ and xYz are considered the same name.
5. A name may **not** duplicate one of the reserved words in the basic vocabulary above.

Unlike Commodore or Apple BASIC, which only looks at the first two characters of a name, **all** characters of a name are "significant" in PROMAL. For example, EXTRAPOLATEX1 and EXTRAPOLATEY1 will be considered as two different variables, even though the first eleven characters are identical.

Similarly, TON is a legal name, even though it contains the reserved word TO (which would make it illegal in BASIC). After compilation, programs using long names **do not** use any more memory or execute any slower than programs with short names, so you should select names which are meaningful. For example, AMOUNT_DUE is probably a better choice for a name than AD.

Some examples of legal names are:

A	ZERO	OldInventory	X_Y_Data
aBc	for4	S___	D_200_
C4	d2000	DearJohn	ET

Some examples of ILLEGAL names (for the reasons indicated) are:

B-4	(second character is not alphanumeric or _)
3D	(first character is not alphabetic)
LIST	(duplicates a reserved word)

Again, remember that you cannot run variable names and PROMAL keywords together the way you can in BASIC. For example,

```
IFID=MEORID=YOU
```

may be an acceptable way to start an IF statement in BASIC, but in PROMAL you would have to write:

```
IF ID=ME OR ID=YOU
```

instead.

DATA TYPES

A data type refers to the **kind** of data that a program can manipulate. PROMAL has four built-in data types, three of which are very simple and quite close to the data types that are used in machine language. This primitive simplicity greatly contributes to PROMAL's speed of execution. The four types are:

Type	Meaning
BYTE	An unsigned integer number between 0 and 255, **or** a single ASCII character, **or** the Boolean value TRUE or FALSE.
WORD	An unsigned integer number between 0 and 65,535.
INT	A signed integer number between -32,767 and +32,767.
REAL	A floating point number between approximately 1.E-37 and 1.E+37. Similar to BASIC's standard numeric data type.

The data type BYTE is a distinguishing characteristic of the PROMAL language. This is very important, because byte variables can be manipulated very rapidly and are frequently needed for the types of applications PROMAL is intended for. As the name implies, a BYTE variable occupies only one byte of memory. WORD and INT (integer) variables each occupy two bytes (16 bits). In memory, the low order 8 bits are stored in the first byte and the high order 8 bits are stored in the next higher address. This is the conventional way to store addresses for the 6502 family processor used in the Apple II and Commodore 64. BYTEs, WORDs, and INTegers **may not** have any fractional part; thus 11 and 12 are okay but 11.5 is not.

REAL variables occupy 6 bytes of memory each. They are similar to the numeric data type used in BASIC (5 bytes each), but are accurate to 11 significant digits instead of 9 significant digits like BASIC. REAL variables have the greatest flexibility because they can store very large and very small numbers, including a decimal fraction. However, they are manipulated **much** more slowly than the other data types (but not as slowly as in BASIC), and therefore should be used with discretion. PROMAL also provides facilities for formatted output, so that you can precisely control the number of digits and number of decimal places printed for REAL output.

BASIC programmers may note the absence of character strings as a standard data type. But **PROMAL can handle strings very well as an array of type BYTE.** String handling is not difficult and will be discussed in detail later.

LITERAL NUMBERS, CHARACTERS, AND STRINGS

Numbers may be written in the usual way. A number written without a decimal point is assumed to be of type BYTE, INT, or WORD, depending on its size and sign. Unsigned values less than 256 are assumed to be BYTE. Larger values are type WORD. Any negative number is assumed to be INT.

Examples of legal BYTE, INT or WORD type numbers are:

```
    0       1       137       22340    65535    -78
```

The following are **illegal** as BYTE, INT or WORD type numbers (for the reasons indicated):

```
    1,333     ; (Cannot have a comma)
    120.6     ; (Cannot have a decimal point - OK for REAL numbers)
    65539     ; (Out of range - must be less than 65536)
```

Literal numbers may also be specified in **hexadecimal**, by using a "**$**" prefix. Hexadecimal (base 16) numbers are often more convenient for specifying memory addresses or bit patterns. For example, it is easier to remember that the Commodore 64 VIC-2 video chip is at address $D000 than at its decimal equivalent, 53248. If you are not familiar with hexadecimal numbers, you may wish to consult your computer's reference manual. Examples of legal hex numbers are:

```
    $0      $a      $2BD      $FFFF      $0012
```


The following are ILLEGAL hex numbers (for the reasons indicated):

```
$1B3.4    ; (Cannot have decimal point in hex number)
FFFF      ; (No $ prefix)
$102B0    ; (Out of range - must be less than $10000).
```

REAL numbers **must be specified with a decimal point**. In BASIC, you can write a real number without a decimal point, but not in PROMAL. If you forget to write the decimal point, PROMAL may accept the number as a valid byte, integer, or word value, without an error indication. However, if you pass this value to a function or procedure that is expecting a REAL value (such as OUTPUT using a #R format), the procedure or function will try to interpret your result as REAL, resulting in a garbage value. Therefore you should **always** be careful to specify a decimal point for a real constant. You may also write REAL literal numbers using the "E" format scientific notation, as in BASIC. Examples of legal REAL numbers are:

```
0.       .0       123.      3.1415926535      -.0000007    56.00
1.2e11   -.003E-10
```

The following are **illegal** real numbers (for the reasons indicated):

```
76000     ; (no decimal point - will be treated as out-of-range integer)
2,333.00  ; (cannot have a comma)
1.21E+50  ; (value out of range; must be less than 1E+37)
```

In specifying literal numbers, you should keep in mind the size limits for the various data types. Only REAL numbers may be larger than 65535 decimal ($FFFF) and may have a fractional part.

PROMAL programs often need to specify single ASCII characters for some operation (**Appendix A** contains a summary of the ASCII character set). To specify a single literal character, enclose it in **single** quotes, for example:

```
'a'       'Q'       '4'       '*'       ' '
```

The PROMAL compiler will substitute the numeric ASCII value of the character. For example, writing 'A' is equivalent to 65 or $41 (see table, **Appendix A**). If you need to show the single quote character itself (') as a literal character, you must double it ('''').

A **literal string** is a group of characters enclosed in **double quotes**. Examples of literal strings are:

```
"A"       "Hello There!"       "26"       "+-*/"       ""
```

When the compiler encounters a literal string in your source program, it generates the ASCII representation of the string, followed by a $00 byte terminator, in your object program. **Literal strings use one byte per character, plus a string terminator which is always a $00 byte.** Therefore the string "A" occupies **two** bytes of memory and the string "Hello There!" occupies 13 bytes in your compiled program. The last example ("") above is called the null string, and contains no characters. This is not the same as a string containing a blank. A blank is a character and occupies space in memory. The 0-byte terminator is always generated automatically by the compiler. A literal string

may contain 0 or more characters. As we will see later, character strings may be up to 254 bytes long, but as a practical matter, a literal character string is limited to the number of characters which will fit on a single line.

The most common use of a literal string is to output a message, which is just as easy as a BASIC PRINT statement:

```
PUT "Hello world!"
```

PUT is actually a built in procedure which should be followed by the address of a string which is to be printed. So for example when the PROMAL compiler sees:

```
PUT "Hello world!"
```

it actually generates a string for you in memory (terminated by a 0 byte), and generates a call to the PUT procedure, passing PUT the address of the string to print. The compiler uses the **address of the first character of the string as the "value" of the string.** If you don´t understand this completely yet, don´t worry about it. The importance and usefulness of this will be explained more fully later.

If you need to include the double-quote character itself (") in a literal string, it should be doubled. For example:

```
PUT  "She said, ""I´ll be back."""
```

will actually cause the program to print:

```
She said, "I´ll be back."
```

You can also embed unprintable codes (such as ASCII control characters or special characters, such as characters to trigger color changes on the Commodore 64) in a string by writing the character **** (**£** pounds sterling key on the Commodore 64, which has no backslash key) followed by **exactly two hex digits** giving the desired character code. For example:

```
PUT "New line \0Dstarting here"
```

will embed a $0D (ASCII carriage return) in mid-string. If you wish to include the **** itself in a string, you should double it, in the same manner as the quote. A particularly useful pair of embedded codes on the C-64 are \12 and \92, which start and stop reverse video output, respectively. On the Apple II, \0F and \0E will enable and disable reverse video.

It is important to remember the difference between a character and a string. A literal character is always a **single** character enclosed in **single quotes**. A literal **string** is zero or more characters enclosed in **double quotes**. This means that ´A´ and "A" do not have the same meaning to the PROMAL compiler. ´A´ occupies a single byte and has the value 65. "A" occupies two bytes, 65 followed by 0, and has the "value" of whatever address the PROMAL compiler assigns to the first character.

Note that you may use PUT only to print characters and strings on the screen. If you need to print the **value** of a variable, you will need to use OUTPUT instead, which is described later.

VARIABLES

PROMAL variables are used to hold values, in much the same way as BASIC variables. However, as we have already seen, PROMAL variables may have long names. PROMAL variables also have a "type" associated with them, which must be BYTE, INT (integer), WORD, or REAL. BASIC variables also have a type, but the type is implied by the name of the variable itself. For example, a % suffix in BASIC indicates an integer type variable and a $ suffix indicates a string type variable. When using PROMAL, however, you must **declare** the type and name of every variable explicitly instead. No special suffixes are used.

DECLARING VARIABLES

In PROMAL programs, all variables **must** be declared before they are used. A variable declaration tells the PROMAL compiler the name of the variable, what type of variable it is, and how much space it will need. A sample variable declaration might be:

```
INT SCORE ;    Game score
```

This declares that you will be using a variable with the name SCORE, and that it will be of type INT. Therefore the variable SCORE will be able to take on signed values between -32767 and +32767. **Only one variable may be declared on a line.** It is considered good programming practice to put a comment after the variable name explaining what it is used for, as is shown above.

In BASIC, you did not have to declare variables (except for the DIM statement, which is a declaration for arrays). Having to list all your variables at the top of the program may seem like a nuisance at first, but you will come to appreciate the value of it. When you pick up a PROMAL program, you can quickly find out the names of all the variables in the program and what they are used for by reading the declarations. If you want to add a new variable, you won't have to search the whole program to make sure the name you choose has not already been used for something else; you just look at the declarations. If you forget to declare a variable before you use it, the PROMAL compiler will flag the variable name with an error message saying "UNDEFINED" when you try to use it.

There is an even more important reason why variables need to be declared. This is best illustrated with an example from BASIC. Suppose you decide to modify an existing BASIC program which uses a variable called X0. You add a few lines to the program, using the variable XO, but the program mysteriously doesn't work. Eventually you discover that the reason is that you typed XO (X-"letter O") but the original variable was X0 (X-"zero"). In this case, BASIC automatically creates a new variable, initialized with a value of zero, instead of using the existing variable X0 which you really wanted. In PROMAL, you would not have this problem because the compiler would flag XO as UNDEFINED. As a matter of historical interest, one of the NASA space program's planetary probes was lost due to a navigational error caused by precisely this kind of bug in a FORTRAN program (like BASIC, you don't have to declare variables in FORTRAN).

As you learn the PROMAL language, you will find other instances like this where PROMAL imposes a certain structure on your programming to help improve the clarity and style of the program.

Unlike BASIC variables, which are automatically initialized to 0, PROMAL does not provide any initialization of variables. This means that you **cannot assume anything about the value of a variable until you have assigned some value to it.** The initial value of a variable is simply whatever happened to be "left over" in the memory location PROMAL assigns to the variable. **Chapter 7** describes a convenient method for initializing all variables to zero with a single statement.

CONSTANT DEFINITION

A constant is a name given to a numeric value which will not change throughout the program. A constant must be defined with a CON statement before it can be used. For example:

```
CON LF=10   ; ASCII linefeed character
```

defines the symbol LF to be 10. After this, anytime the PROMAL compiler encounters the name LF, it will substitute the value 10 instead. There are two differences between constants and variables. First, the value of the constant is permanent and is associated with the constant name at compile time. Second, no memory is set aside to save the value of the constant in the data area. Instead, any time the constant is referenced, the compiler generates the value of the constant (in the same manner as a literal constant) in the executable code of the program. Only one constant can be defined on a line. Again, it is considered good practice to add a comment to a constant definition explaining what the constant is. If you are an assembly language programmer, you may recognize that a PROMAL constant is equivalent to an assembly language "equate". You may also define the type of the constant explicitly, for example:

```
CON WORD STARTLOC=$40
```

defines STARTLOC to be of type WORD with a value of 40 hexadecimal. If you don't specify the type explicitly, PROMAL will assume type BYTE if the value is less than $100, INT if it has a minus sign, and type WORD otherwise. Later we will learn more about constant definitions, after we learn about operators and expressions.

You may **not** declare a REAL constant. Instead, you should use a DATA statement if you wish to associate a name with a permanent value of type REAL. Disallowing REAL constants saves memory and reduces the complexity of the compiler.

ARRAY VARIABLES

PROMAL allows arrays of any of the four data types, with up to eight subscripts. Subscripts for the array are enclosed in square brackets "[ ]", not in parentheses like BASIC. This makes it easy to tell the difference between an array element and a function call (where parentheses are used to enclose the arguments, as will be discussed later). Like all other variables, arrays must be declared before they can be used. An array variable declaration is similar to a simple variable declaration, but is followed by the number of elements of the array desired. For example:

```
BYTE BUFFER [81]
```

declares an array of type BYTE which can hold 81 elements (BUFFER[0] through BUFFER[80]). It is important to observe that if you define an array as X[N], then **the last element is X[N-1], not X[N]**, because X[0] is the first element.

It is considered good programming practice to define a constant which controls the size of a subsequently declared array. This will usually make it easier to alter the program later. For example:

```
CON BUFSIZE = 100
BYTE BUFFER1 [BUFSIZE]
BYTE BUFFER2 [BUFSIZE]
...
```

You may not use a variable as the dimension for an array, however. This is because the PROMAL compiler allocates memory for the array at compile time; the size of the array must be known at compile time, not when the program is actually run.

The **subscripts for an array of any type must always be of type WORD.** If an array subscript evaluates to type BYTE, it will be "promoted" to WORD automatically. A subscript which evaluates to type REAL will cause the compiler to generate an error message. The maximum subscript which can be used is dependent on the amount of free memory. **When you refer to an array name without subscripts (or backets), the address of the array will be used.** The importance of this will be illustrated later.

It is possible to define both simple variables and arrays at specified locations in memory. For example you can define the screen memory as an array starting at $0400 (1024). This is kind of declaration is called an **external** variable, described in Chapter 6, "Interfacing".

CAUTION: When array elements are referenced, PROMAL **does not perform any bounds checking** (because of the adverse affect on performance). Therefore a sequence like:

```
WORD I
BYTE BUF[10]
BYTE LINE[8]
...
I=12
...
BUF[I]=0
```

will not produce an error message and will move the 0 into part of the LINE array instead of the BUF array as was intended. You should always take care to insure that array indices stay in bounds, or strange and invariably unpleasant results will occur!

Multiple dimension arrays have the subscripts separated by commas. For example:

```
BYTE SCREENIMAGE [80,25]
REAL STIFFNESS [10,20,3]
...
IF SCREENIMAGE [0,I] = ' '
  SCREENIMAGE [0,I] = SCREENIMAGE [1,I]
...
BEND = TORQUE * STIFFNESS[I,J,K]
...
```

The amount of memory required to store an array is the product of its declared dimensions times the size of each element. The SCREENIMAGE array above uses 2000 bytes, and the STIFFNESS array uses 3,600 bytes. Multiple dimension arrays are mapped into memory such that incrementing the first subscript will address elements that are physically adjacent in memory. Or, another way to visualize this is to say that SCREENIMAGE is organized as 25 groups of 80 bytes each (**not** 80 groups of 25 bytes each). Therefore if you wish to have a two dimensional array of text, the column subscript should come first and the row subscript second, as was done for SCREENIMAGE above. This is discussed further in **Chapter 7.**

DATA DEFINITION

A data definition is similar to a variable but has a predefined initial value which is determined at compile time. For example:

```
DATA REAL PI = 3.1415926535
```

defines a data item of type REAL which will be predefined to the value of PI.

Unlike constants, data definitions can define arrays as well as simple variables. The DATA definition is most frequently used to define a table of values which will not be changed by the program. The DATA definition looks similar to a variable declaration, except that it starts with the word DATA and is followed by an "=" and the desired value (or values). For example:

```
DATA BYTE MYTABLE [] = 23, 12, 8, 4, 2, 1, 0
```

This line defines an array called MYTABLE of type BYTE having 7 elements. Notice that the size of the array is not given in the brackets; the PROMAL compiler counts the number of elements for you. You must explicitly define the value of all elements. The first element of the array will be MYTABLE[0] and will be initialized to 23. The last element will be MYTABLE[6] and have the value 0.

You may **not** define multiple-dimension DATA arrays. Only a single dimension is permitted for DATA declarations.

You **may not** change the value to a data item with an assignment statement. If a data name appears on the left side of an assignment statement, the compiler will generate a "Variable Expected" error. It is **possible** to force the data items to be altered with an assignment statement to a variable array which overlaps the data items, but this is considered poor programming practice (and will also cause your program to be reloaded from disk if you try to re-execute it, because data items are included in the checksum which the EXECUTIVE uses to determine if a program has been corrupted).

If you wish to use a table of data items to set the initial values of a variable array which will subsequently be altered, the correct procedure is to copy the data array to another variable array (using the BLKMOV procedure, described later), and then alter the variable array.

The data definition is the one statement in PROMAL which can consist of multiple lines. In order to continue the data definition on additional lines, either the = sign or a comma should be the last character of the preceding line. For example:

```
DATA WORD LIST [] =
0,45,13,27,
0,46,13,28,
1,46,14,28,
1,47,14,29
```

defines an array of 16 words.

DATA statements are frequently used to define an array of strings which can be used for messages, etc. during the program. For example:

```
DATA WORD ERRORMSG [] =
  "Function Successful.",                   ; 0
  "Illegal widgit.",                        ; 1
  "Widgit not found.",                      ; 2
  "You must specify a Widgit Number first." ; 3
```

This statement defines a table of four words, each initialized to point to a string. Later in your program, if you wanted to print the "Widgit not found." error message, you could simply write:

```
PUT ERRORMSG[2]
```

PUT is a built-in LIBRARY procedure which displays the string specified, in this case the third string in the table.

Please note that the type of the above data array is WORD, not BYTE. This is because each element of the array is a string. You may recall from our discussion of strings that the "value" of a string is the **address** of its first character; therefore a WORD is necessary to hold this address.

OPERATORS

An **operator** is a special symbol which indicates an action to be performed. PROMAL provides the following operators:

Op.	Description	Example	Result
+	Addition	3 + 5	8
-	Subtraction or negation	48 - 11	37
*	Multiplication	-10.32 * .034	-.35088
/	Division (fraction discarded except REAL)	200 / 30	6
%	Remainder (mod)	200 % 30	20
<<	Left shift	7 << 1	14
>>	Right shift	$A0 >> 4	$0A
<	Relational operator less than	4 < 9	TRUE
<=	Relational operator less than or equal	6 <= 6	TRUE
<>	Relational operator not equal	'A'<>'A'	FALSE
=	Relational operator equal	'A'=65	TRUE
>=	Relational operator greater than or equal	10 >= 'a'	FALSE
>	Relational operator greater than	3 > 8	FALSE
AND	Logical AND operator	3>1 AND 4<10	TRUE
OR	Logical OR operator	2<=1 OR 8>9	FALSE
XOR	Logical exclusive OR	$00 XOR $FF	$FF
NOT	Logical complement	NOT TRUE	FALSE
#	Address of variable	#X	addr of X
:<	Extract low byte of WORD or INT	$1234:<	$34
:>	Extract high byte of WORD or INT	$1234:>	$12
:+	Convert to WORD	$5A:+	$005A
:-	Convert to INT	$FF:-	+255
:.	Convert to REAL	45:.	45.0
@<	Indirect through pointer to BYTE	PTR@<	see text
@-	Indirect through pointer to INT	PTR @-	see text
@+	Indirect through pointer to WORD	(PTR+2)@+	see text
@.	Indirect through pointer to REAL	PTR @.	see text

Some of these operators may look familiar from your experience with BASIC; others are entirely new. These operators may be combined with operands, which may be numbers, characters, strings, constants, variables, data, or functions, to produce expressions. We shall now examine the most important of these operators in detail.

ARITHMETIC EXPRESSIONS

Like BASIC, arithmetic expressions are evaluated from left to right (in the absence of parentheses), with multiplication and division having a higher priority than addition and subtraction. Therefore the expression:

```
3 + 4 * 5
```

evaluates as 23, not 35. A summary of operator precedence is given below.

OPERATOR PRECEDENCE

(operators in the same row have equal precendece)

```
:<, :>, :+, :-, :., @<, @+, @-, @., #        Highest precedence
NOT
*, /, %, <<, >>
- (negative)
+, -
<, <=, <>, =, >=, >
AND OR XOR                                    Lowest precedence
```

The arithmetic operators, +, -, *, and /, work in the expected fashion, but with a few twists. First of all, remember that PROMAL deals with integers (whole numbers) as well as real numbers. The result of arithmetic on type BYTE, WORD or INT cannot have a fractional result. Therefore 5 / 2 evaluates as 2, not 2.5 (any fraction is always discarded). However, 5. / 2. evaluates as 2.5, because the presence of the decimal point tells the PROMAL compiler that the numbers are REAL.

Note for **Commodore 64**: Be careful not to type the shifted "+" character on the keyboard when you want a plus sign. It looks like a plus sign, but isn´t (the same applies to BASIC).

Most operators take two operands. For most operators, these two operands do not have to be of the same type. In a mixed expression involving operands of different types, the operands are usually "promoted" to the "higher" type automatically, where BYTE is the "lowest" and REAL is the "highest" type. The table below summarizes the results of a partially evaluated expression of the type shown in the left column when an operator is encountered with a new operand of the type shown in the top row:

RESULT TYPE FOR MIXED MODE EXPRESSIONS

Next operand involved is...

Present Type is...	BYTE	WORD	INT	REAL	
BYTE	BYTE	WORD	INT	REAL	Result type
WORD	WORD	WORD	INT	REAL	Result type
INT	INT	INT	INT	REAL	Result type
REAL	REAL	REAL	REAL	REAL	Result type

The TYPE of the data being operated on must be considered. For example, adding two variables of type BYTE will always result in a value which is also of type BYTE, even if the result is too large to fit in a BYTE variable. For example, if X is a variable of type BYTE which has been previously assigned the value of 254, then the expression X+4 will NOT have a value of 258, but 2. This is because BYTE variables can only take on values between 0 and 255, so that when you add 4 to 254, the result is (258-256) = 2.

If you don't quite understand this, think of PROMAL BYTE, INT and WORD variables as being like the odometer on your car. Most odometers go up to 99,999.9. If your odometer reads 99,998.0 and you drive 4 more miles, the odometer will read 00,002.0, not 102,000.0. A PROMAL variable of type BYTE only goes up to 255 ($FF hex), and then "wraps around" again starting at 0. A numeric expression which overflows the maximum value representable simply "wraps around" like this with no indication of an error. Similarly, if you subtract a larger BYTE operand from a smaller BYTE operand, the result is "wrapped around" but still positive. For example, 3 - 4 evaluates to 255 (think of what happens if you turned back the odometer 4 miles when it had a reading of 3).

Since by definition a BYTE type variable is unsigned, you cannot apply the negation operator to it directly, so the byte is automatically promoted to type INT (integer) before the negation is performed. This "promotion" is only done in the temporary work area called the accumulator where PROMAL does its arithmetic; it does not change the type or size of the original variable.

An operand of type WORD also is always positive, but in this case the largest possible "odometer reading" is 65535 (FFFF hex). For example if Y is a variable of type WORD with a value of 1, then Y-3 is 65534 ($FFFE), not -2.

Only integers and reals may take on negative values. To understand how integers work, again consider your auto odometer. If you started out at 0 and turned the odometer back 1 mile it would read 99,999.0. Turn it back another mile and it would read 99,9998.0. If you wanted to use your odometer to measure both forward and backward movement from 0, you might define everything from 0 to 49,999.9 as positive, and everything from 50,000.0 and above as negative, effectively splitting the total number of representable numbers in two (half positive and half negative). This is exactly how INT variables work in PROMAL.

In two bytes there are 65,536 possible numbers, which we divide in two, with 0 to 32767 being considered positive ($0000 to $7FFF). The other half of the numbers represent negative numbers, with -1 represented by $FFFF. The most negative number possible is -32768, or $8000. However, since there is no +32768 number representable, the number -32768 is disallowed. This number scheme is called "two's complement" arithmetic, and is standard on almost all computers.

For example, consider the following fragment of a PROMAL program:

```
BYTE X
WORD Y
WORD ANSWER
...
X = 254
Y = 300
ANSWER = X + Y
```

This will produce the expected result of ANSWER=554. However, if you change the last line to read:

```
ANSWER = X + 3 + Y
```

then the result will be ANSWER=301, because X and 3 are both type BYTE, so X + 3 evaluates to 1; this is then promoted to a word and added to Y to give 301. If the order of the operands was changed to:

```
ANSWER = X + Y + 3
```

then the result would be 557, because X + Y would be evaluated first, with X being promoted to WORD before making the addition.

Most of the time you will not have to worry about mixing different types in an expression, but when you do you should bear in mind the order of evaluation. You can "force" an operand to be promoted (or "demoted") from one type to another with the "type cast" operators, which are:

```
:<   Extract low order byte from word or integer (or convert real to byte).
:>   Extract high order byte from word or integer.
:+   Convert to word (unsigned).
:-   Convert to integer (signed)
:.   Convert to real (floating point).
```

These operators are written immediately **after** the operand which they are to change. For example:

```
ANSWER = X:+ + 3 + Y
```

would result in ANSWER=557, because the :+ operator will promote or "cast" X to a word before performing the addition with Y. The expression X:+ is read as "X cast to a word".

There are four special cases for arithmetic operators.

1. The % operator (remainder) cannot be applied to REAL operands. The sign of the result is always considered positive for the % operator.
2. If you multiply or divide two operands of type BYTE, both operands will be promoted to WORD, and the result will be type WORD.
3. Taking the negative of a BYTE or WORD converts to an INT. No error is given if the result is out of range (result truncated to 16 bits).
4. Dividing by zero will produce a fatal run-time error. A "zero divide" error can be triggered by any of the following:

   a. Division by 0 (X / 0).
   b. Remainder by 0 (X % 0).
   c. A REAL result larger than the largest representable value (about 1.E+37).
   d. Conversion of a REAL to a BYTE, WORD, or INT which cannot be represented (e.g., 100000. :+).

RELATIONAL OPERATORS

The relational operators (<, <=, <>, =, >=, >) are the same as their BASIC counterparts, and return a value of TRUE or FALSE. In PROMAL, TRUE is represented by a byte of value 1 and FALSE by a byte of value 0. For purposes of comparison in a conditional statement such as an IF statement (which we will study later), **any non-zero value is considered TRUE.** The result of a comparison using a relational operator is always type BYTE. Promotion of operands in a comparison is the same as for the arithmetic operators, but the result is always type BYTE.

The fact that the result of a relational operation can be interpreted as 0 or 1 as well as FALSE or TRUE can be useful. For example, the two statements:

```
IF PHASORS > 100
  SCORE = SCORE + 1
```

can be replaced by the single equivalent statement:

```
SCORE = SCORE + (PHASORS > 100)
```

because the expression (PHASORS > 100) will evaluate as 1 if TRUE and 0 otherwise.

The relational operators all have equal priority of evaluation and are of lower priority than any arithmetic operators, so that "normal" comparisons will produce the expected result when written without parentheses. For example the expression:

```
3 * 3 > 3 + 3
```

evaluates as TRUE (1).

Please note that you may **not** compare two strings by simply using the relational operators on the variables involved, because this would merely compare the **addresses** of the strings, which has no relation to the **content** of the strings. To compare strings, use the CMPSTR function, described in the LIBRARY MANUAL.

LOGICAL OPERATORS

The logical operators (AND, OR, NOT, XOR) may be combined with relational operators or used for bit-by-bit Boolean operations. These operators may only be used on operands of type BYTE, which is normal if using them in conjunction with relational operators. All logical operators have an equal priority of evaluation which is lower than the arithmetic and relational operators, so that "normal" combinations of operators will produce the expected result without parentheses. For example the expression:

```
X > 100 AND  Y = 0
```

is equivalent to:

```
(X > 100) AND (Y = 0)
```

and will evaluate TRUE if X is greater than 100 and Y is 0.

AND, OR and XOR are useful in performing bit-by-bit Boolean operations and masking operations (on type BYTE operands only). For example:

```
PORT  AND  $0F
```

will "mask off" the high order 4 bits of PORT. As you may have already discovered, these masking operations are frequently needed to manipulate selected bits within a byte.

The operator NOT is a unary operator which converts any non-zero byte to 0, and 0 to 1. To perform a bit-by-bit complement, use XOR $FF instead.

SHIFT OPERATORS

The operators << and >> perform left and right shifts, respectively. The operand to be shifted appears on the left side of the operator, and the shift count on the right, for example:

```
XVAL << 4
```

shifts the value of XVAL left by four bits. Shifts may be applied to all data types except REAL; however, **the shift count must be of type BYTE.** The shift count should be in the range of 0-8 for BYTE operands and 0-16 for WORD or INT operands. Shift operators have the same precedence of evaluation as multiplication and division. One of the most frequent uses of shifts is to perform multiplications or divisions by powers of 2. For example:

```
COUNT << 3
```

will compute eight times the value of COUNT much faster than:

```
COUNT * 8
```

Right shifting by N is equivalent to (and much faster than) dividing by 2 to the Nth power. Shifts are also sometimes used in conjunction with the logical operators for manipulating data into specific bits of a register. Bits shifted out of a byte or word are lost; 0 bits are always shifted into the word (even if it is a negative integer). The result of a shift on type INT is type WORD. There is no built-in operator to perform bit rotations.

INDIRECT AND ADDRESS OPERATORS

The operator **#** is the **address** operator. It can only be applied to a variable or data name (not to a number, string, constant or function). The # operator returns the address of the variable which follows it. For example:

```
WORD PTR
REAL STRENGTH
...
PTR = #STRENGTH
```

sets the variable PTR to the address of the variable STRENGTH in memory. The # operator can also be used to find the address of a particular element in an array, for example:

```
WORD PTR
DATA BYTE COMDCHAR [] = 'D','X','P','A','E','Q'
...
PTR = #COMDCHAR[2]
```

will set PTR to the address of the character 'P'.

The operators @<, @-, @+ and @. are **indirect operators.** They are used to access data "pointed to" by some variable or expression. The expression to the left of the indirect operator should be of type WORD. If it is of type BYTE, it will be promoted to type WORD automatically. For example:

```
WORD POINTER
REAL VALUE[10]
...
POINTER = #VALUE[7]
...
IF POINTER@. > 0.5
  ...
```

Here POINTER is set to the address of a certain element of an array of REALs. Later, the expression POINTER@. can be used to test the value of that element. The expression "POINTER@." can be thought of as "the real number pointed to by POINTER."

One of the most common uses of the indirect operators is to extract characters from strings. For example, consider the following program fragment:

```
BYTE BUFFER [80]
WORD PTR
BYTE CHAR
...
PTR= BUFFER
...
CHAR = PTR @<
...
```

This sequence will set CHAR to the first character of the array BUFFER. Although this could also have been done with the more straightforward statement:

```
CHAR = BUFFER[0]
```

the use of PTR allows more versatility, since PTR could point to **any** array, not just the BUFFER array. Pointers and indirect operators are very useful in passing arrays and strings to subroutines to be operated on, as you will see in Chapters 5 and 7.

Note that you **may not** use the indirect operators to identify the destination variable for an assignment statement. Therefore

```
PTR@< = 10  ; ILLEGAL!
```

is **not legal.** You may use the predefined array M, which is defined in the Library as an array of bytes encompassing all of memory, to solve this problem. The above example could be correctly written as:

```
M[PTR] = 10  ; Right!
```

The use of pointers and the array M is discussed further in the section on subroutines and in Chapter 5 and 7.

GLOBAL VARIABLES

Variables are normally declared first in your program, before the executable statements. These variables are called **global** variables, because they can be accessed from anywhere in your program. Later another kind of variable will be introduced called a **local** variable. Local variables are defined **inside** subroutines, and are known only inside that subroutine. Global variables are defined before any subroutines (or between subroutines), and are known everywhere thereafter in the entire program, (including inside all subroutines). This distinction will be clarified in Chapter 5, where subroutines are discussed.

Now that you know how to declare variables and form expressions, you are ready to learn how to bring these pieces together with the reserved words to form statements, and then combine these statements into a complete working program.

CHAPTER 4: STATEMENTS

INTRODUCTION

In this chapter you will learn about PROMAL language statements. If you have only programmed in BASIC and not in another "high-level" language you should study this chapter very carefully. If you have programmed in Pascal or "C" this chapter will be important for understanding the differences as well as the similarities of PROMAL and other "structured" languages.

Some PROMAL statements are similar to statements in BASIC. For example,

```
XV = YV + 17
```

is an assignment statement, which is very similar to a BASIC LET statement. However, there are some important differences between BASIC statements and PROMAL statements, including:

1. Statements **do not** have line numbers.
2. Only **one** statement is permitted on a line.
3. A statement may not occupy more than one line (with the exception of the DATA statement).
4. Keywords and variables **must** be separated from each other by blanks or other punctuation marks as required by the statement.

SYNTAX DIAGRAMS

In many ways, PROMAL allows you a great deal more flexibility in constructing statements than BASIC. In order to help you determine exactly what makes up a legal statement, a set of syntax diagrams is included in **Appendix P.** These syntax diagrams tell you graphically how to construct a legal PROMAL statement. Syntax diagrams are not difficult to use, once you are familiar with them. If in the following descriptions you are unsure about a PROMAL statement's correct syntax, you may refer to the diagrams in **Appendix P**, and the accompanying discussion of how to read them.

PROGRAM STATEMENT

Every PROMAL program must start with a PROGRAM statement of the form:

PROGRAM Name [**OWN** [**EXPORT**]]
-or-
OVERLAY Name [**EXPORT**]

where **Name** is a legal PROMAL identifier not used for any other purpose. The PROGRAM line declares the command name by which you will execute the program when it is loaded into memory. You should always **make the PROGRAM name the same as the file name you COMPILE.** The OWN keyword is optional, and is normally not used. If specified, it will cause the compiled program to be loaded into memory with the global variables allocated immediately after the program, rather than being shared with other programs in high memory. This and the EXPORT and OVERLAY keywords are discussed further in Chapter 8 and in the optional Developer's Guide.

ASSIGNMENT STATEMENT

The assignment statement is the simplest and most fundamental statement in PROMAL (or in any other language). You are familiar with it in BASIC. Its form is:

```
variable = expression
```

where expression can be a constant, a variable, a function, or a combination of these in an arithmetic or relational expression. See the Syntax Diagrams in **Appendix P** for all the possibilities. The assignment statement assigns the contents of (or results of) the expression on the right side of the "=" sign to the variable on the left side.

The variable on the left cannot be a DATA item. Here are some sample assignment statements:

```
X=0
ENDPAGE = TRUE
SMALLX =MIN(X1,X2,X3)
VAL[I]=3.14159*RADIUS[I]*RADIUS[I]
YBIGGER = Y > X AND Y > Z
```

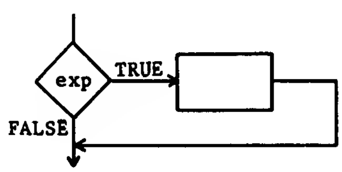
CONDITIONAL STATEMENTS

A conditional statement is a statement which alters the order of execution of statements based on evaluating a condition. In BASIC, the conditional statements are IF, FOR...NEXT, ON...GOTO, and ON...GOSUB. PROMAL has conditional statements which are more powerful and easier to read and understand than the related BASIC statements. The PROMAL conditional statements are the IF, WHILE, REPEAT, FOR, and CHOOSE statements.

IF STATEMENT

By far the most common conditional statement is the IF statement. It can take several forms. The simplest form is:

```
IF expression
  statement 1
  statement 2
  ...
statement n
```



In this form, the expression is tested, and if it is TRUE, then **all** the indented statements following it are executed. If it is FALSE, then control passes directly to statement n, on the same level of indentation as the IF.

For example:

```
IF X > 10
  OLDX = X
  X = 10
Z=X
```

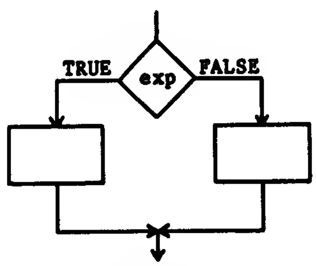
In this case, the conditional expression tests if X is greater than 10. If so, then OLDX is set to X and X is set to 10. If not, the two statements after the IF statement are skipped. In either event, the line Z = X is always executed.

BASIC programmers, please note that you **cannot** put THEN, GOTO, or anything else on the same line as the IF, after the condition! You **must** put the statements to be executed on the lines after the IF, and they **must be indented.** The indentation must be exactly two columns to the right of the IF. The proper indentation is easily obtained by using the TAB key (or CTRL I) in the PROMAL EDITOR.

If you have ever taken any courses in programming, you probably were told that indentation is a good way to show a program's structure. PROMAL simply enforces this concept. The indentation **does** show the structure of the program. **This is probably the most important feature of the PROMAL language.** By using indentation as a syntactical element of the language, PROMAL is able to do away with a host of confusing statement delimiters and begin-end brackets which pervade other structured languages.If you don't indent, you'll get an error message when you compile your program.

A second form of the IF statement has an ELSE clause:

```
IF expression
  statement 1
  ...
ELSE
  statement 2
  ...
statement n
```



In this form, the indented statements after the IF are executed if the expression is TRUE, and the statements after the ELSE are executed otherwise. This form is used to select one of two mutually exclusive paths. For example:

```
IF X > 100
  POINTS = 3
ELSE
  POINTS = 1
SCORE = SCORE + POINTS
```

If X is greater than 100, POINTS is set to 3 and control passes to the last line. If X is not greater than 100, POINTS is set to 1 and control passes to the last line.

The final form of the IF statement has one or more ELSE IF clauses before the final ELSE, for example:

```
IF CHAR = 'D'
  DRAW
ELSE IF CHAR = 'E'
  ERASE
ELSE IF CHAR = 'Q'
  EXIT
ELSE
  OUTPUT "ILLEGAL COMMAND."
```

This form is used to choose one of a number of mutually exclusive paths. Please note that the **only** thing that can follow an ELSE on the same line is an IF and a condition. The ELSE without an IF must be the last ELSE associated with the initial IF. Also be sure that ELSE and IF are typed as two words, not one.

IF statements may be "nested" to any depth needed. For example:

```
IF X > 100            ; 1
  IF Y > X            ; 2
    Z=3+X             ; 3
    Y=0               ; 4
  ELSE                ; 5
    Y=1               ; 6
    IF X > 200        ; 7
      Z=Y-100         ; 8
Q=Y+Z                 ; 9
```

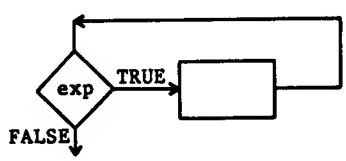
In this example, each IF controls all the statements with greater indentation. For example, if the first IF (statement 1) is false, then control will pass directly to statement 9. If statement 1 is true, then statement 2 decides if statements 3 and 4 should be executed or skipped. The only way statement 8 will ever be executed is if statement 1 is true, statement 2 is false, and statement 7 is true. You should have no doubt about which IF statement an ELSE "belongs to"; it is always the one with the same indentation.

Indentation plays a key role in making programs readable. You will soon be able to just scan over a PROMAL program or subroutine and immediately be able to understand its logic. Since PROMAL does not have a GOTO statement, there will be no mystery as to how you get to a certain statement. By just looking at the indentation, you will have a "picture" of the program organization.

WHILE STATEMENT

Next to IF, WHILE is the most commonly needed control statement in PROMAL. It has the following form:

```
WHILE expression
  statement 1
  ...
statement n
```



The WHILE statement evaluates the conditional expression. If it is TRUE, the indented statements are executed, as in an IF statement. After the last indented statement is executed, control returns to the WHILE statement and the condition is re-tested. The loop is repeated until the expression evaluates as FALSE; control then passes to **statement n**, which starts in the same column as the WHILE statement. The indented statements in a WHILE loop may be executed zero or more times. For example:

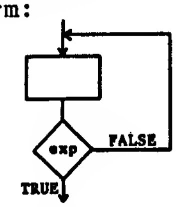
```
SUM = 0
X=0
WHILE X < XLIMIT
  SUM = SUM + X
  X = X + 1
Z=X
```

This program fragment forms the sum of the integers from 0 to XLIMIT. At the end of the loop, Z will be equal to XLIMIT.

REPEAT STATEMENT

A REPEAT statement is very similar to a WHILE statement, except that the condition is tested at the end of the loop instead of the top. The REPEAT statement has the following form:

```
REPEAT
  statement 1
  ...
UNTIL expression
```



The indented statements are executed one or more times. After the first execution of the indented statements, the conditional expression is evaluated. If the result is FALSE, control passes back to the top of the loop. If the statement is TRUE, control passes to the next statement after the UNTIL. For example:

```
REPEAT
  CHAR = GETC
UNTIL CHAR = 'A'
```

GETC is a standard LIBRARY function which returns a key from the keyboard. Therefore this loop waits for an 'A' to be typed, ignoring all other input.

FOR STATEMENT

The FOR statement is similar to a BASIC FOR-NEXT loop, but is more restrictive. A FOR loop has the form:

```
FOR Iter = Low TO Hi
  statement 1
  ...
statement n
```

Iter **must be a variable of type WORD,** and Low and Hi must be expressions which evaluate to the lower and upper bounds for the loop. For example:

```
WORD BUFFER [100]
WORD I
...
FOR I = 0 TO 99
  BUFFER [I] = 0
```

will initialize the array BUFFER to 0. Note that the iteration variable must be a simple variable, not an array element or expression, and must be type WORD. Also note that the loop must iterate upward, not downward (as is permitted in BASIC). There is no "STEP" size option as in BASIC; the step size is always 1. The indented block of a FOR loop is always executed at least once, even if Low is greater than Hi. These restrictions allow the FOR loop to execute very rapidly. If you need a FOR loop which doesn´t meet these requirements, use a WHILE loop instead.

CHOOSE STATEMENT

The CHOOSE statement is a multi-way branch, somewhat similar to BASIC´s ON-GOSUB statement, or the CASE statement of Pascal. It has the following form:

```
CHOOSE expression
choice 1
  statement 1
  ...
choice 2
  statement i
  ...
...
ELSE
  statement j
  ...
statement k
```

The CHOOSE statement works like a multiple-choice test. The expression is evaluated, and each of the choices listed below is compared to it in succession. When a match is found, the indented statements are executed. If no match is found, the indented statements after the ELSE are executed (think of the ELSE as "none of the above"). In any event, control always winds up at statement k, the first non-indented line after the ELSE. For example:

```
CHOOSE GETC
´B´
  X=0
  START
´C´
  CONTIN
´L´
  X=9999
  LASTLINE
ELSE
  PUT "Illegal key letter"
X=1
```

The program fragment above inputs a character from the keyboard (function GETC). If the character is ´B´, then X is set to 0 and the START subroutine is called, and control transfers to the last line (X=1). If the character is ´C´, the CONTIN subroutine is called instead, and control then passes to the last line. If the character does not match any of the choices, then an error message is output (the PUT does this), and control passes to the last line.

The choices for the CHOOSE must match the type of the expression in the CHOOSE line **exactly**, and the expression must **not** be type REAL. Note that this means if you have a CHOOSE with an expression of type WORD or INT, and your choices are small BYTE constants such as 0, 2, 100, etc., you must remember to promote the choices to type WORD or INT.

WRONG!	RIGHT
WORD NUM	**WORD** NUM
...	...
CHOOSE NUM	CHOOSE NUM
1	1:+
PROCESS_1	PROCESS_1
2	2:+
PROCESS_2	PROCESS_2
ELSE	ELSE
PROCESS_OTHER	PROCESS_OTHER

Any CHOOSE statement can be simulated with an IF statement with an appropriate number of ELSE IF clauses. However, CHOOSE will often be more efficient since you do not have to spell out each comparison explicitly.

The CHOOSE statement also has an alternative form where the word CHOOSE appears alone, for example:

```
CHOOSE
CHAR < ' '
  CONTROLCHAR
CHAR > $7F
  ILLEGALCHAR
ELSE
  NORMALCHAR
```

In this form, each of the choices is evaluated in succession until one evaluates TRUE. If all of the choices are FALSE, then the indented statements after the ELSE are executed. This is exactly equivalent to an IF with several ELSE IF clauses, except you do not have to write the ELSE IF's explicitly.

BASIC users should note that after the indented statements are executed for one of the choices, control automatically passes to the first non-indented statement after the ELSE; you do not need to put a GOTO after each like you do for a BASIC ON-GOSUB. Also note that the ELSE is mandatory, because it indicates the final choice ("none of the above").

BREAK STATEMENT

Sometimes it is desirable to "break out" of a loop at a point other than where the conditional test is done. The BREAK statement provides this capability for WHILE and REPEAT loops (but **not** for FOR loops!). For example:


```
WHILE TRUE    ; (do forever)
  IF I@< = ´(´
    IF (I+1)@< >= ´a´ AND (I+1)@< <= ´z´
      IF (I+2)@< = ´)´
        BREAK
  I=I+1
FOUND=I
```

This program segment will search all of memory for a single lower case alphabetic character enclosed in parentheses. Executing BREAK causes control to immediately pass to the statement after the **end** of the most recent WHILE loop (i.e., to the FOUND=I statement).

NEXT STATEMENT

The NEXT statement is used to cause an immediate jump to the **top** of the current WHILE or REPEAT loop (but **not** a FOR loop). For example:

```
INQUOTE=FALSE

COUNT=0
REPEAT
  CHAR = GETC
  IF CHAR=´"´
    INQUOTE=NOT INQUOTE
    NEXT
  IF INQUOTE
    NEXT
  COUNT=COUNT+1
UNTIL CHAR=CR
```

This program segment counts the number of characters typed up to the next carriage return, excluding characters enclosed in quotes (including carriage returns in quotes). The NEXT statements pass control back to the top of the loop so as to ignore characters in between (and including) quotes. There may be better and easier ways to do this -- this is just for illustration.

NOTHING STATEMENT

The NOTHING statement does not perform any action, and the PROMAL COMPILER does not generate any object code for a NOTHING statement. This may seem of dubious merit, but is actually useful. For example:

```
REPEAT
  NOTHING
UNTIL GETC = CR
```

This loop simply waits for a carriage return from the keyboard, ignoring all other characters. The NOTHING statement fulfills the syntactical requirement that at least one indented statement must follow the REPEAT, but it performs no action. If you tried to leave out the NOTHING statement, you would get an error message from the compiler.

SHORTCUTS FOR CONDITIONAL STATEMENTS

You may recall that TRUE is represented by a byte with value 1 and FALSE by a byte with value 0. Several "shortcuts" can be used to take advantage of this fact to generate faster executing PROMAL statements. First of all, for a variable FLAG of type BYTE,

```
IF FLAG=TRUE
```

can be written equivalently but more economically as:

```
IF FLAG
```

Also the sequence:

```
IF X > 100
  FLAG = TRUE
ELSE
  FLAG = FALSE
```

can be more economically written as:

```
FLAG = X > 100
```

ESCAPE AND REFUGE STATEMENTS

The ESCAPE statement and REFUGE statement are unique to PROMAL and do not have a counterpart in other structured languages or BASIC. PROMAL, like many modern structured languages, does not have a GOTO statement, which results in much cleaner, more readable and more bug-free programs. There are occasions when you might wish you had a GOTO. This is best illustrated by an example.

Suppose you had a complex application program, with many layers of subroutines. Suppose further that at some low-level subroutine you come to a point where you need a piece of logic that could be paraphrased as:

```
IF Disaster
  Print error message
  Exit back up to the top level routine.
```

This is a common problem. Unfortunately, other languages do not provide a way to "exit back up to the top level routine". Instead, you must "unwind" all the CALLs with RETURNS. In other structured languages, you typically "solve" this problem by testing some global "Disaster" flag after returning from a lower level subroutine to short-circuit further processing, for example:

```
LOWERSUB    ; call lower subroutine
IF DISASTER ; if had problem in LOWERSUB
  RETURN    ; don't go any further
```

Each higher level subroutine would perform the same logic, until you "unwind" all the way back up to the desired routine. While this method works, it is unwieldy and dilutes the performance and clarity of the program with a lot of duplicate error checking.

PROMAL solves this problem a different way. The REFUGE statement can be thought of as an "executable label", and the ESCAPE statement as a GOTO which can exit back to a previously executed REFUGE.

The syntax of the ESCAPE and REFUGE statements is:

```
REFUGE n
ESCAPE n
```

where n is a constant between 0 and 2, allowing up to 3 different "refuges" to be defined concurrently in a single program. (Note: actually, there is also a REFUGE 3, but this is reserved for a special purpose and is described in the optional DEVELOPER'S GUIDE). Executing:

```
REFUGE 2
```

defines the statement after it as refuge number 2. Subsequently executing:

```
ESCAPE 2
```

will cause an immediate re-entry into the last subroutine (or main program) executing a REFUGE 2 at the line after the REFUGE statement, and will restore the context of the subroutine at that point. By restoring context, we mean that all intermediate variables, return addresses, etc., which would normally be "pending" when a RETURN is executed are discarded, up to the point where the refuge was executed. An ESCAPE is somewhat like a NEXT or BREAK statement, except that instead of just jumping to the beginning or end of a loop, you can **jump to anywhere you've been before**. It is the programmer's responsibility to insure that you do not try to ESCAPE to a REFUGE in a routine that has already returned (which will leave control in no-man's land!). On the next page is an example of fragments of a program using a REFUGE and ESCAPE:

```
PROC ERROR ; print error message and escape
 ARG WORD ERRNO
 ...
 BEGIN
 PUT NL, ERRORMSG[ERRNO],NL  ;display message
 ESCAPE 1
 END

 PROC CHECKCHAR
 BEGIN
 IF CHAR <> LEGAL
   ERROR 3
 ...
 END

 PROC PROCESSWORD
 ...
 CHECKCHAR
 END

 PROC DOPHRASE
 ...
 PROCESSWORD
 END

 PROC DOLINE
 ...
 REFUGE 1  ;Come here after error
 WHILE GETL(LINE)
   DOPHRASE
 ...
 END
```

CHAPTER 5: PROCEDURES AND FUNCTIONS

PROMAL provides a greatly enhanced subroutine capability compared with BASIC. Some of the most important characteristics of PROMAL subroutines are:

1. Subroutines may be either **PROC**edures or **FUNC**tions. Functions return a value which may be used in an expression. Procedures do not return a value.

2. Both procedures and functions must be defined (or "declared") before they can be called.

3. Functions and procedures are called by merely referencing their name in a statement.

4. Both procedures and functions may be passed **ARG**uments which they may operate on.

5. Both procedures and functions may have **local variables** which are known only within the scope of the subroutine. These local variables may duplicate other names outside the subroutine without interference.

6. Procedures and functions may be called recursively.

Let us now explain these concepts and show how to make effective use of subroutines.

BUILT-IN FUNCTIONS AND PROCEDURES

PROMAL does not have any built-in statements to do input and output, like BASIC PRINT and INPUT statements. Instead, PROMAL relies on a **LIBRARY** of pre-defined subroutines and functions to provide input and output. These routines are always resident in memory, and are used by the EDITOR, EXECUTIVE, and COMPILER as well as programs you write. When you use these subroutines, they could easily be mistaken for a special statement. For example:

```
OUTPUT "Hello World!"
```

appears just like a statement. There is no "CALL" or "GOSUB" keyword to reveal that this is really a subroutine call, with a passed argument of "Hello World!". This is no accident. A design intent of PROMAL is that subroutines should give you much of the power of adding your own statements to the language. You call subroutines of your own in the same way.

The built-in subroutines are described in detail in the LIBRARY MANUAL. At this point we would like to introduce you to just the most important of these routines, so that you can perform basic input and output operations.

Before you can call a subroutine, you must define it. For the LIBRARY subroutines, this is done by having the following statement near the top of your program:

```
INCLUDE LIBRARY
```

This defines all the standard LIBRARY routines to the PROMAL compiler. For now, it is sufficient for you to know that this LIBRARY gives the name and location of each of the built-in routines. You can display the Library with a **TYPE L** command from the EXECUTIVE.

SIMPLE OUTPUT

Probably the most fundamental of the standard procedures is called **PUT.** It outputs **single characters or strings** to the screen. It can have one or more arguments. For example:

```
PUT "Hello world!",NL
```

This statement calls the PUT procedure and passes it two arguments to be displayed. The first argument is the string "Hello world!", and the second argument is NL, the pre-defined "newline" character (which is the ASCII control character CR and has the value 13 for the Apple/Commodore version of PROMAL). Unlike a BASIC PRINT statement, **you must explicitly output an NL each time you want to start a new line.** This makes it easy to build up a composite line with several calls to PUT.

Please note that, unlike BASIC, you **cannot** print the numeric value of a variable with the PUT statement. You can only **print strings or characters.** To print a numeric value, you will want to use the OUTPUT procedure.

FORMATTED AND NUMERIC OUTPUT

OUTPUT is a procedure for performing formatted output to the screen. It accepts one or more arguments. **The first argument must be a string.** It is called a **format string,** because it tells the format in which any additional arguments should be printed. If you have ever used a BASIC version which supports PRINT USING, OUTPUT is similar. Actually, it is most similar to the PRINTF function in the C language.

The format string contains text to be printed on the screen as well as formatting information. The special lead-in character **#** is used to start a **field specification** (sometimes called a field descriptor), which tells **how** to print something. For example:

```
INT SECS
...
SECS = 673
OUTPUT "The answer is #I seconds.", SECS
```

These statements will display:

```
The answer is 673 seconds.
```

The value of the argument SECS replaces the format field specification #I. The "#I" indicates that the second argument should be displayed as an integer. The most commonly needed field specifications include:

#I	Print the argument as a signed integer number.
#W	Print the argument as an unsigned number (not for REAL variables!)
#H	Print the argument as a hexadecimal number.
#S	Print the argument as a string.
#C	Print a carriage return.
#E	Print the REAL argument in scientific notation.
#R	Print the REAL argument with a decimal point.

The OUTPUT statement can have more than two arguments. The format string **must have a field specification for each argument to be printed.** For example:

```
WORD N
...
N = 257
OUTPUT "#C#W decimal = #H hexadecimal.",N,N
```

will display:

```
257 decimal = 101 hexadecimal.
```

after a carriage return. Notice that the #C does not go with any argument; it just prints a carriage return.

You can also specify a "field width" in the format string (for example, to make columns of numbers line up). These options are fully described in the LIBRARY MANUAL. For REAL output, you normally specify both a field width and the number of decimal places to be displayed, in the form:

```
#w.dR
```

where w is the field width (from 3 to 12 characters), and d is the desired number of decimal places. For example:

```
REAL BUCKS
...
BUCKS = 276.10
OUTPUT "$#7.2R",BUCKS
```

will display:

```
$ 276.10
```

whereas BASIC would always print $ 276.1 instead.

SIMPLE INPUT

Now that you know how to output to the screen, let's see how you input from the keyboard. The procedure GETL is used to get one line from the keyboard and store it in an array **as a string.** GETL allows all editing features (backspace, insert, delete, CTRL-B, etc.) allowed by the EXECUTIVE. It returns when the RETURN key is pressed. For example:

```
BYTE LINE[81]
...
GETL LINE
```

reads one line from the keyboard and puts it in array LINE. After the call, the LINE array will be terminated by a 00 byte. It will not include the carriage return. Normally only one argument is present for the GETL procedure, and that argument is the **address** of where to put the line. Remember that **the name of an array without a subscript evaluates as the address of the array.** Optionally, you may include a second argument which is the maximum line length to accept (excluding the 0 byte terminator). For example:

```
GETL LINE, 20
```

will read a line from the keyboard up to 20 characters long. Additional characters on the line will be ignored. If the second argument is not specified for GETL, a maximum of 80 characters can be input.

The GETL statement is much more powerful than a BASIC INPUT statement because GETL supports a complete set of line-editing keys, as shown in **Table 1** of the USERS MANUAL. These keys are consistent with the editing keys used in the PROMAL EXECUTIVE and EDITOR.

One of the most useful features of GETL is the ability to recall prior lines by pressing CTRL-B. Another powerful feature for many applications is the use of function keys to "call up" pre-defined strings of up to 31 characters (much like many commercial "keyboard enhancers"). The LIBRARY MANUAL describes how to use FKEYSET to define a string to be substituted for a function key.

NUMERIC INPUT

How do you read in a numeric value from the keyboard? This is not quite as simple in PROMAL as in BASIC, because PROMAL does not have a built-in statement to read a number. Instead, you do it in two parts. First, you read a line into a buffer as described above. Then you convert the value represented by the string using function STRVAL or STRREAL. STRVAL converts a string to the numeric value it represents of type INT, or WORD. STRREAL is used to convert type REAL numbers. It is similar to the BASIC function VAL. For example, to read a number called HEIGHT from the keyboard, you could write:

```
BYTE BUF [81]
WORD HEIGHT
BYTE INDEX
...
GETL BUF
INDEX = STRVAL(BUF,#HEIGHT)
...
```

The STRVAL function expects at least two arguments. The first argument is the address of the string to be converted. The second argument is the **address** of the variable to receive the value. To specify the address of the variable (rather than the value), you need to specify the # operator, as shown above. Forgetting the # in front of the variable is a common error that results in the value being installed at whatever address is the current value of HEIGHT, so be careful! Also remember that the destination variable must be type WORD or INT,

not BYTE. Besides installing the value of the number into HEIGHT, function STRVAL will return an **index** of type BYTE. This index indicates the number of characters which were scanned in the string before the end of the number. If the INDEX is returned as 0, it indicates that no numeric digits were entered, probably representing an error condition. For example, if you typed

```
123
```

then INDEX would be returned as 3 and HEIGHT as 123. This method may seem a little ungainly and roundabout at first, but it allows a great deal of flexibility and programmer-defined error recovery, which is essential for serious programming. STRVAL also supports hexadecimal input, formatted input, and variable numbers of inputs on a line. These options are described in the LIBRARY MANUAL.

BASIC users accustomed to using the INPUT statement to prompt for a numeric input from the keyboard and input it may want to incorporate the following general purpose PROMAL routine. This INPUTR function will give a prompt for input and return the REAL value that the user enters from the keyboard. If an illegal input is entered from the keyboard, it repeats the prompt.

```
FUNC REAL INPUTR ; Prompt
  ; Prompt for numeric input from keyboard, return one REAL value.
ARG WORD PROMPT  ; Desired prompt
REAL TEMP        ; Value to be returned
BYTE INDEX       ; Index to # chars scanned
OWN BYTE BUF[21] ; Temp buffer for typed input line
BEGIN
REPEAT
  PUT NL,PROMPT  ; Display prompt
  GETL BUF,20    ; Get typed input
  INDEX=STRREAL(BUF,#TEMP)
  IF INDEX=0     ; No legal digits?
    PUT NL,"Please enter a numeric value"
UNTIL INDEX > 0
RETURN TEMP      ; Return value typed in
END
```

A sample program fragment using this routine for input might look like this:

```
...
REAL HEIGHT
REAL WIDTH
REAL AREA
...
HEIGHT=INPUTR ("Height of triangle? ")
WIDTH = INPUTR( "Base of triangle? ")
AREA=0.5*HEIGHT*WIDTH
OUTPUT "#CArea of triangle is #12.4R square units.#C", AREA
...
```


An example in the STRVAL section of the LIBRARY MANUAL contains a variation of the routine above for entering WORD or INT data instead of REAL values. For your convenience, both these functions are provided on disk as source files INPUTR.S and INPUTW.S, so you can easily include them in your programs or modify them to suit your individual needs.

The LIBRARY contains many more Input-Output routines, including file input and output. We will postpone a discussion of these routines until later.

USER-DEFINED SUBROUTINES

When you define your own PROMAL subroutine, you write it in the following general form:

{header}
{arguments}
{local variables}
BEGIN
{body}
END

The {header} is a single line that identifies the start of the subroutine. It has the form:

PROC name

or

FUNC Type Name

which defines whether the subroutine will be a procedure or a function. For example:

```
PROC SORT
```

declares the start of a procedure called SORT.

```
FUNC BYTE TESTPORT
```

declares the start of a function called TESTPORT which will return a value of type BYTE. The type returned may be BYTE, WORD, INT, or REAL.

The **{arguments}** and **{local variables}** will be discussed very shortly.

The **{body}** part of the procedure or function is contained between the **BEGIN** and **END** statements. It contains the executable statements of the procedure or function. When program control reaches the END statement, the subroutine will return to the calling program. Optionally, the RETURN statement can be used to return before the END statement.

For FUNCtions, a RETURN statement is required and **must** be followed by an expression which evaluates to the value to be returned by the function. For example:

```
RETURN YVAL+1
```

will return the value of YVAL+1 as the value of the function. Function values will be covered in more detail shortly.

PASSED ARGUMENTS

A powerful feature of PROMAL is the ability to use arguments passed to procedures and functions. BASIC does not support passed arguments (except in a very limited sense in simple function definitions using FNx, which is rarely used). To use passed arguments, the PROMAL subroutine definition should include one argument declaration line for each argument which is to be passed to the subroutine. An argument declaration looks like a simple variable declaration, with the word ARG in front:

ARG Type Name

Type is the desired data type which may be BYTE, INT, WORD, or REAL. Name is the desired name of the subroutine, formed in the same way as other variable names.

For example:

```
PROC SORT
ARG WORD N
ARG WORD PTR
```

declares two passed arguments, N and PTR, both of type WORD. The **order in which the arguments are declared is the same as the order in which the corresponding values will be passed.** For example, if the SORT procedure above was called with:

```
BYTE ARRAY[100]
...
SORT 26, ARRAY
```

then when SORT begins executing, N will have the value of 26 and PTR will have the address of ARRAY. As you can see, a procedure is called by simply writing the name of the procedure to be called. Arguments are passed by putting the arguments after the procedure name. Each argument can be an expression, and arguments are separated by commas. When you call a procedure or function which you have defined, the number of arguments must agree exactly with the number you declared, or you will get an error message from the compiler. The initial value of the arguments depends entirely on the values passed.

If the routine is later called with:

```
SORT CURSIZE+1, BUFFER
```

then N will have the value CURSIZE+1 and PTR will have the value BUFFER. As you might imagine, this substitution process makes subroutines very versatile.

A very important fact about arguments is that the names declared for passed arguments are **local** to the subroutine. This means that the name declared has meaning only within the subroutine where it is declared. It may duplicate a name used outside the subroutine for another purpose without harm. Technically, we say that the **scope** of the variable is local to the subroutine. This means that you can write a subroutine for one program and later copy it into another program without having to worry if the names you chose for argument variables will "collide" with some other variable names already in use. For example, suppose the following program fragment calls our sample routine:

```
N=11
SIZE=17
SORT SIZE,BUF
Z=N
```

After the SORT routine returns, what will be the value of N when it is assigned to Z? Will it still be 11 or will it be 17 because SIZE is 17 and was substituted for N in the SORT routine? The answer is that N will still be 11, because the N in the subroutine is only meaningful within the subroutine where it is declared.

PROMAL passes arguments on a "call by value" basis, with arguments passed on the microprocessor's hardware stack. This means that when you pass an argument to PROMAL, the argument is evaluated and this value is substituted for the local variable. Therefore, if the local variable's value is altered within the subroutine, it will not affect the value in the calling routine. For example, suppose that part of our SORT routine looks like this:

```
PROC SORT
ARG WORD N
ARG WORD PTR
BEGIN
...
N=0
```

Assuming we call the subroutine with:

```
SIZE=17
SORT SIZE, BUFFER
```

What will be the value of SIZE when the subroutine returns? Will it still be 17 or will it be 0? It will be 17, because the variable N is local to the SORT routine, and contains a copy of the value of SIZE, not the SIZE variable itself.

A passed argument need not have the same type as the type declared for the variable in the subroutine, although in general it is good practice to make them the same. If you pass a BYTE argument to a variable declared to be a WORD, the value will be converted to a WORD as it is passed. For example:

```
BYTE SIZE
...
SIZE=10
SORT SIZE, BUFFER
```


will work properly, even though SIZE is type BYTE and will be substituted for N inside the subroutine, which has a declared type of WORD. Technically, the declared variable is sometimes called a "formal parameter" and the value passed in the call to the subroutine is called an "actual parameter".

Although a BYTE actual parameter may be passed to a routine with a WORD formal parameter, you should be very careful to **only** pass a REAL argument to a REAL formal parameter. Passing REAL variables to a routine expecting BYTE, INT or WORD arguments will produce very strange results!

Sometimes you may want to modify a global variable which is passed as an argument to a subroutine. In this case, the usual procedure is to **pass the address** of the variable to be changed to the routine, and let the routine set the value using the globally pre-defined array **M**, which is defined in the library to be an array of bytes encompassing all of memory. For example, our subroutine SORT may wish to sort the array of bytes pointed to by PTR. To set the first value of this array to 0, for instance, we could write in the body of our subroutine:

```
M[PTR]=0
```

Arguments must be declared as simple (unsubscripted) variables. **You may not declare an argument which is an array.** This does not mean that you can't **access** a global array from inside a subroutine. You may do this freely. You cannot **declare** the array inside the subroutine. It is also possible for a subroutine to operate on an array whose address is passed as an argument. In our example procedure, SORT, the array BUFFER was given as the second argument. Remember that when an array name is used without a subscript, PROMAL generates the address of that array. Therefore our call will pass the address of the array to the subroutine, which is why PTR is declared to be a WORD. Since PTR contains the address of the start of the array, elements of the array can be accessed using the indirect operators, or by the **M** array as shown above. For example:

```
BYTE BUFFER[10]
...
PROC SORT
ARG WORD N
ARG WORD PTR
BEGIN
...
IF PTR@< > (PTR+1)@<
...
END

...
SORT 10,BUFFER
...
```

The IF statement above will compare the value of the first and second bytes of the array BUFFER.

Installing a REAL value into a variable whose address is passed as an argument can be accomplished by a block move of exactly 6 bytes from the address of the local variable to the desired destination. See BLKMOV in the LIBRARY MANUAL for information on block moves.

LOCAL VARIABLES

A local variable is similar to an argument, but has **no** initial value defined. Local variables are known only within the subroutine in which they are declared, and "disappear" when the routine returns. Local variables are most often used for temporary storage within the routine. Local variables should be declared **after** the last ARGument in the procedure or function. A local variable declaration appears the same as a simple global variable declaration. For example:

```
PROC SORT
ARG WORD N
ARG WORD PTR
WORD I
BYTE CHAR
...
```

declares two local variables, I and CHAR. The compiler knows these are local variables and not global variables because they are declared within the subroutine. Except for having no initial value, local variables behave identically to arguments. In particular, they are allocated on a stack and therefore must be simple variables, **not arrays**.

To illustrate the concepts of global and local variables, here is a complete, simple function which returns the number of blanks in a string. Remember that a PROMAL string is an array of bytes terminated by a $00 byte.

```
FUNC BYTE NUMBLANKS  ; string
;   return # blanks in string
ARG WORD STRINGPTR   ;address of string
BYTE N               ;counter
BEGIN
N=0
WHILE STRINGPTR@<
  IF STRINGPTR@< =' '
    N=N+1
  STRINGPTR=STRINGPTR+1
RETURN N
END
```

There are a number of important concepts here. First, the line

```
WHILE STRINGPTR@<
```

will evaluate TRUE as long as the byte pointed to by STRINGPTR is not 0; that is, not end-of-string. Second, the line

```
STRINGPTR=STRINGPTR+1
```

is perfectly legal and does not change the address of the original string passed to the subroutine. STRINGPTR is local to the subroutine and is initialized to point to the start of the string, and can be used to step through the string one character at a time.

Finally, notice that the variable N, which is used to count the number of blanks, must be initialized to 0 explicitly because PROMAL does not initialize local variables to anything. The result of the function is returned via the RETURN N statement.

If you call this function with the statement:

```
NB = NUMBLANKS("Hello there everybody!")
```

then NB will be set to two. Notice that functions, unlike procedures, must be called with the arguments enclosed in parentheses. This is because a function can be part of a larger expression, for example:

```
GETL MYMSG
NBP1 = NUMBLANKS(MYMSG) + 1
```

will set NBP1 to the number of blanks in MYMSG plus 1.

OWN VARIABLES

Local variables may not be arrays, and the value associated with a local variable "disappears" upon exit from the subroutine in which it is defined (because space for the variable is allocated on a stack). This meets the requirements of the vast majority of variables in subroutines. Sometimes though, you may want to have a variable known only within the subroutine, but which is an array or needs to preserve its value from call to call. This can be done by declaring an OWN variable. For example, the statements:

```
OWN BYTE TEMPBUF[8]
OWN WORD COUNT
```

declare two variables whose names are local to the subroutine in which they are declared, but which will maintain their values through multiple subroutine invocations. The most common use of OWN variables is to provide a scratch array needed for intermediate processing by a subroutine. OWN variables should be declared **after** all arguments and local variables, but before the BEGIN statement in a subroutine.

GOOD PROGRAMMING PRACTICE WITH SUBROUTINES

It is considered good programming practice to add a comment after the header line of a procedure or function definition which tells the function of the subroutine, what it expects for input, what it returns for output, etc. Many PROMAL programmers like to put a comment at the end of the header line listing the required arguments. If you do this, it will be easy to refer to the header line for a quick reminder of what arguments are expected. Finally, it is a good idea to put a comment on each argument declaration and local variable, identifying the purpose of the variable and any constraints on its use.

You should make frequent use of procedures. It is generally best to keep procedures short. Most PROMAL routines should have only a few lines of code. The PROMAL COMPILER, EDITOR, and EXECUTIVE contain hundreds of subroutines with less than twenty lines of code. If you have a procedure of more than about 50 to 100 lines, it should probably be broken up into lower-level subroutines. It is often a good idea to use procedures for various phases of processing, even if the procedures are only called in one place in the entire program. PROMAL subroutine calls require very little overhead time. Therefore in general you need not worry about extra procedure calls slowing down your program the way GOSUBS slow down BASIC. Remember, PROMAL doesn't have to search for your subroutines the way BASIC does. The PROMAL compiler generates the address of the routine during compilation, so calls are very fast, and take the same amount of time no matter where in the program the subroutine definition is located.

Subroutines are named the same way as variables. It is often a good idea to pick a verb which describes the main action of the subroutine for its name. Then when you call the subroutine with one or more arguments, the statement will be very readable, for example:

```
DISPLAY SPACESHIP
```

calls procedure DISPLAY with an argument of SPACESHIP.

You can learn a lot about procedures and functions by studying the sample programs on the PROMAL diskette.

RECURSION

PROMAL fully supports recursion. This means that it is permissable for a procedure or function to call itself, or for procedure A to call procedure B which in turn calls procedure A again. This capability is very important in certain programming disciplines, such as writing compilers, artificial intelligence applications, and in symbolic math. It is also possible to have forward references to procedures and functions. Techniques for recursive programming are described in **Appendix J**.

The ability to perform recursion on the Commodore 64 and Apple II is limited by the architecture of the 6502 processor, which only has a 256 byte stack. Although PROMAL has been carefully written to work around this limitation as much as is practical, you should not expect too many levels of nesting (or recursion) before you get a STACK ERROR message. You will use up more stack space as the number of local variables or passed arguments increases. A typical function with one passed argument and one local variable can call itself about 40 times before stack overflow occurs. This is why it is possible to get a stack overflow error while compiling a program with an expression that is very complicated and uses many levels of parentheses. The compiler uses recursion extensively to parse statements and can run out of stack space as repeated recursive subroutine calls are made to process complex statements.

USING THE INCLUDE STATEMENT FOR MULTIPLE SOURCE FILES

For large programs, it is not practical to edit the entire program at once. Instead, you should break up your source program into several files. Your main file can then have an INCLUDE statement for each of the sub-files. You have already seen how the INCLUDE statement is used to include the LIBRARY definitions in your program. You can do the same thing for your own programs. For example, the statement:

INCLUDE FILESUBS

will cause the compiler to pause at this point in the main file and compile all the lines in the file FILESUBS.S before continuing.

You may put an INCLUDE statement anywhere you can put a declaration. A ".S" extension will be assumed for the file name if one is not specified. The INCLUDE file can have a drive or directory prefix. For the Commodore 64, due to limitations of the Commodore 1541 disk drive, you cannot have nested include files (that is, a file INCLUDEd in the compilation cannot itself contain another INCLUDE statement). For the Apple II, INCLUDE files may be nested up to 3 deep. However, you may need to specify more than three buffers for ProDOS (see the BUFFERS command in the PROMAL USER'S GUIDE) in order to use nested INCLUDEs. INCLUDE statements can also be used to import definitions from separately compiled modules, as is described later in the LOADer section.

ENABLING AND DISABLING LISTING OUTPUT WITH THE LIST STATEMENT

The L option on the COMPILE command is used to enable listing output for the compiler. When making a listing, you can also disable the listing for parts of your program with the LIST statement. The LIST statement can appear anywhere a declaration can appear. It can have either of the following forms:

LIST Constant

or

LIST

The first form enables the listing if **Constant** evaluates to a non-zero value and disables the listing if it is zero. The second form restores the listing mode to whatever it was prior to the previous LIST (on or off). This form is useful at the end of a subroutine package which has the listing turned off, where it is not known if you will want the listing ON or OFF after the end of the subroutine package.

You may have any number of LIST statements in a program. If the L option is not specified on the COMPILE command, no listing will be made regardless of any embedded LIST statements. For an example, if you **TYPE L**, you will see how the listing of the LIBRARY is disabled.

CONDITIONAL COMPILATION

Sometimes you may have several versions of a program which vary only slightly. For example, the PROMAL EXECUTIVE is slightly different for the COMMODORE 64 version and the APPLE II version. In cases like this, you may wish to take advantage of PROMAL's **conditional compilation** capability. Conditional compilation allows you to generate several versions of a program from a single source file (perhaps with INCLUDEs), by specifying which version you wish to compile on the COMPILE command line. Here's how it works.

Inside your program source file, you can "bracket" the source lines which should only be compiled for a certain version. This is done by inserting a line above the first version-dependent line, containing a question mark in column one followed immediately by a single character representing the version for which the following lines are to be assembled. For example, you might choose 'A' for an Apple-dependent portion and 'C' for a Commodore-dependent section:

```
PROGRAM MYPROG
INCLUDE LIBRARY
...
?A
PUT NL,"The COLOR command is not supported on the Apple."
?

?C
COLOR=NUMVAL
?
...
```

In this example, there are two conditionally compiled blocks, each of a single line. The first block is started by the ?A and is only intended to be compiled if we want an Apple version. The ? by itself (exactly in column 1) terminates this block. The second block is started by ?C and terminated by the second plain ? character.

Selecting which (or neither) block should be compiled is selected by the V (version) option on the COMPILE command. For example,

```
COMPILE MYPROG V=A
```

will cause the Apple version to be compiled, and

```
COMPILE MYPROG V=C
```

will cause the statement COLOR=NUMVAL to be compiled instead. If you don't specify either V=C or V=A on the command line, then <u>neither</u> block will be compiled.

Conditional compilation is sort of like a simple IF statement, except that if the conditional block is skipped, the compiler <u>does not generate any code at all</u> for those statements; the result is equivalent to removing them with the EDITOR (or, more precisely, to "commenting them out" by putting a semi-colon in front of each).

You can specify a conditional block for either or two or more versions. For example:

```
?AC
```

starts a conditional block which will compile if either V=A or V=C is specified, but won't otherwise.

If no V option is specified, the compiler will compile a block which starts with

```
?*
```

if it appears. This is useful for embedding an error message to remind the user that a version must be specified on the command line. For example:

```
PROGRAM MYPROG
?*
*** YOU MUST SPECIFY V=A OR V=C TO COMPILE THIS PROGRAM! ***
?
INCLUDE LIBRARY
...
```

If you compile this program without the V option specified, the compiler will attempt to compile the warning line, giving an error message and displaying the line. If you specify a V option, the warning will not be compiled.

You may have any number of conditional blocks in a program. However, you may not nest conditional compilation blocks (that is, you can't have a conditional block inside another conditional block). The size of a conditional block is arbitrary, and may span INCLUDE files. It is your responsibility to insure that each block is terminated by a ? in column 1. You may have any number of single character version indicators following the ? character which begins a conditional block, but you may specify only one character on the V command line option (if you specify more, all but the first character will be ignored, so V=APPLE is equivalent to V=A).

The RELOCATE.S file on the PROMAL SYSTEM DISK illustrates the use of conditional compilation.

CHAPTER 6: INTERFACING

This chapter describes how your PROMAL program interfaces with its environment, including:

1. Disk files.
2. The printer.
2. The EXECUTIVE.
3. Your Apple II or Commodore 64 computer hardware.

FILES AND DEVICES

The PROMAL USERS GUIDE contains a section on the requirements for naming and using PROMAL files and devices on your computer. You may wish to review this material, particularly **Table 3** and **Table 4**, before proceeding with this section, which describes how to input and output to files and devices from within your PROMAL program. In particular, please remember that PROMAL file names normally have at least two characters, while device names have a single character. Also remember that the system will normally assume a default file extension of ".C" for file names if no file extension is specified.

PROMAL provides functions and procedures in the LIBRARY to input and output to files and devices. **The same routines may be used to access a file or device (such as the printer).**

OPENING AND CLOSING FILES

Before a file or device can be accessed, it must be **opened.** The library function OPEN performs this task. The OPEN function returns a **file handle** (sometimes called a file descriptor), which is a pointer to a table maintained in memory by PROMAL, used to control file I/O. This file handle should be assigned to a variable of type WORD. Once the file is open, the file handle can be used to direct subsequent I/O to the file desired. For example:

```
WORD INFILE
...
INFILE = OPEN("MYFILE.D",'R')
```

opens file MYFILE.D for reading. The second argument must be of type BYTE (not string!) and indicates the mode of operation, chosen from the following:

'R' (or omitted)	Open the file for read access.
'W'	Open a new file for write access.
'A'	Open an existing file for append access.
'B' (Not available on Commodore)	Open for both read and write access.

If the **file handle is returned as 0, then the open was <u>not</u> successful,** and an error code is available in a globally predefined variable called IOERROR. IOERROR will be one of the following:

IOERROR	Meaning (if file handle returned as 0)
0	No error, normal result
1	Illegal mode character
2	Illegal file or device name
3	Disk drive is not ready (or wrong volume name for ProDOS)
4	File not found
5	File already exists (for 'W' access attempt)
6	No free channels or buffers (too many open files)
7	Attempt to write on write-protected disk ('W' or 'A' access)
Other	Other error, see Commodore 64 disk manual or Apple ProDOS manual.

You should always test for an open failure before attempting I/O to the file or device. For example:

```
WORD INFILE    ;file handle for input file
...
INFILE = OPEN ("MYFILE.D",'R')
IF INFILE = 0
  IF IOERROR=4    ;the most likely error
    ABORT "MYFILE.D file not found."
  ABORT "#CDisk OPEN error #W",IOERROR
; Open was successful...
...
```

You can also open the devices for input or output in the same way, for example:

```
INFILE = OPEN ("W",'R')
```

opens the Workspace for read access. Recall that the Workspace is a single in-memory file with a fixed maximum size (variable size for the Commodore 64).

```
WORD PRINTER
...
PRINTER=OPEN("P",'W')
IF PRINTER=0
  PUT NL,"CANT OUTPUT TO PRINTER"
...
```

opens the printer device for output and checks the file handle for a successful open.

FUNCTIONS FOR FILE AND DEVICE I/O

Probably the most commonly used routines for accessing files are GETLF and PUTF. Function GETLF gets a line of text from a file or device, and procedure PUTF outputs characters or strings to a file or device. The first argument for all file-access routines **must** be the file handle of the previously-opened file or device. Function GETLF returns TRUE if it successfully got a line, and FALSE if end-of-file was encountered immediately. For example:

```
WHILE GETLF(INFILE, BUFFER)
  ...
```

reads a line from the file opened successfully with file handle INFILE and installs a line into the array BUFFER, which is assumed to have been previously declared as an array of bytes. The WHILE statement is frequently used in conjunction with this function to continue reading until end-of-file. The body of the WHILE loop contains whatever processing is to be done on the line. You may specify an optional third argument on function GETLF which specifies the maximum number of characters to be returned from the line. GETLF should only be used to read text files, not compiled programs.

Procedure PUTF is similar to the screen-output routine, PUT, except that the first argument must specify the file handle of a successfully-opened file or device. For example:

```
WORD OUTFILE
...
OUTFILE = OPEN ("MYFILE.T", 'W')
IF OUTFILE = 0
  PUT "Unable to open MYFILE.T for output."
  ABORT
PUTF OUTFILE,"This line goes to MYFILE.T",NL
PUTF OUTFILE,"So does this.",NL
...
```

Like PUT, PUTF can contain any number of strings or single character arguments to be output. It will not output a carriage return unless explicitly indicated. PUTF can put any kind of data byte out to a file, not just printable characters.

The OUTPUTF procedure is equivalent to the OUTPUT procedure for formatted output, except that the first argument must be the desired file handle. For example:

```
OUTPUTF OUTFILE, "#C#H   #S",LINENUM,LINE
```

outputs to the file previously opened.

Other functions are available for single character and block input-output to files and devices. These are described in the LIBRARY MANUAL.

STDIN AND STDOUT FILE HANDLES

When your PROMAL program begins, you already have two open file handles available for use. These are the globally predefined WORD variables **STDIN** and **STDOUT**. By default, these file handles normally point to the keyboard and screen, respectively. However, they can be **redirected** to any file or device when your program is executed by an EXECUTIVE command (See the MEET PROMAL and PROMAL USER'S MANUAL for details). Therefore if you input from STDIN and output to STDOUT, your program's output will be redirectable under EXECUTIVE control. For example:

```
PUTF STDOUT,"This goes to the screen or where I redirect it.", NL
PUTF STDOUT,"So does this."
```

You do not have to open STDIN or STDOUT. These variables already hold open file handles when your program starts. If you want some output to go to the screen no matter what, you simply use PUT and OUTPUT instead of PUTF and OUTPUTF. This will bypass I/O redirection set by the EXECUTIVE.

OUTPUT TO PRINTER

To output to the printer, simply OPEN the "P" device for output and use the file output procedures with the for the printer specified as the first argument. For example:

```
WORD PRT     ; Handle for printer
REAL X
...
PRT=OPEN("P",'W')    ; Open printer for writing
IF PRT = 0
  ABORT"#cUnable to open printer."
...
PUTF PRT, NL, "This line goes to the printer."
...
X=124.35
OUTPUTF PRT, "#cThe answer is #12.4R",X
...
```

Note that just because you were able to OPEN the printer successfully does not necessarily mean the printer is ready to receive output. If you OPEN the printer, send it output, and the system appears to hang, it may be that the printer is not on-line or ready to print.

For the **Commodore 64**, you must remember not to power on the printer while DYNODISK is on (also, for interfaces such as the CARDCO, the interface must be off too; for the CARDCO interface, this means that the single wire must be unplugged from the back of the computer while DYNODISK is on). You can turn DYNODISK on and off from within a program, if desired (see **Appendix G**).

When doing output to a printer, be sure to send a NL after the last line, since many printers keep the line in their internal memory until a CR is received to cause them to print.

PRINTER CONTROL

Printers vary considerably in terms of interface to the computer. To help reduce the difficulty in dealing with various printers and printer interfaces, PROMAL pre-defines several variables (in file PROSYS.S) to govern printer output.

For the **Apple II**, you can control whether or not PROMAL should automatically send a LF after every CR to your printer. See **APPENDIX E** for details. Also, if your computer is a IIc or is connected by a **serial** interface, you will need to set another variable to perform graphics or escape sequences. This is also described in **APPENDIX E**.

For the **Commodore 64**, printers (or printer interfaces) often have special modes selected on the basis of the "secondary address". The following three variables can be used to control your printer:

```
EXT ASM BYTE C64PSA AT $0DF3  ; Desired secondary address (default 7)
EXT ASM BYTE C64PUL AT $0DF4  ; Bit 7=1=flip case (default=$80=yes)
EXT ASM BYTE C64PDV AT $0DF5  ; C-64 printer device # (default 4)
```

These variables can be set by your program before opening the "P" device, or by direct commands from the EXECUTIVE or your BOOTSCRIPT.J file. See **APPENDIX E** for details.

PRINTER ESCAPE SEQUENCES

Most printers use ASCII control characters or escape sequences to select different attributes such as underlining, font selection, character size, etc. It is very easy to send these sequences to the printer using PROMAL PUTF statements, after you have set up your printer control options properly as described above. For example, if your printer manual tells you that the particular escape sequence you want is:

Escape sequence	Decimal form	BASIC form
ESC W 1	27, 87, 49	LPRINT CHR$(27);CHR$(87);CHR$(49)

then in PROMAL you could just write:

```
PUTF PRT, 27, 87, 49
```

assuming you have previously opened the "P" device with handle PRT.

For **Commodore 64** computers using interfaces such as the CARDCO, you may have to select some special mode before sending escape sequences to your printer. For example, the CARDCO model G+ needs to be opened with C64PSA=5 and C64PUL=0 (as described above) in order to select "transparent mode".

OUTPUT TO SCREEN AND PRINTER

Sometimes you may want to output the same text to the screen and the printer. This can be accomplished by executing the same PUTF or OUTPUTF statement twice, using different file handles. For example, the following program fragment supports selective output to either just the screen or to the screen and printer:

```
WORD SP [2]  ; File handles for screen, printer
WORD BOTHOUT ; =0 if just screen, 1 if screen + printer output wanted
WORD I
...
SP[0]=STDOUT ; screen file handle (already open)
BOTHOUT=0
PUT NL,"Do you wish output to printer too?"
IF TOUPPER(GETC)='Y'  ; yes?
  SP[1]=OPEN("P",'W') ; then open printer for writing
  BOTHOUT=1
```

```
...
FOR I=0 TO BOTHOUT
  PUTF SP[I],NL,"This will go to printer & screen if BOTHOUT=1",NL
...
```

ARGUMENT PASSING FROM THE EXECUTIVE

PROMAL provides a simple mechanism for passing command-line arguments from the EXECUTIVE to a program. The standard LIBRARY defines two globally predefined variables which are preset by the EXECUTIVE before control is passed to a program:

NCARG	is the number of arguments passed to the program.
CARG[1]	is a string containing the first argument, if present
CARG[2]	is a string containing the second argument, if present
...	
CARG[NCARG]	is the last argument.
CARG[0]	is a string containing the command which was executed (the command name)

For example, if your program is executed by the EXECUTIVE command:

```
DOIT  Myfile  2367
```

then on entry to the program,

```
NCARG        will be 2
CARG[1]      will be "MYFILE"
CARG[2]      will be "2367"
CARG[0]      will be "DOIT"
```

All the CARG array elements will be pointers to strings containing the arguments. The program should consider these strings as DATA and not modify them in place.

The EXECUTIVE normally treats blanks as the delimiters between arguments. Both leading and trailing blanks are stripped off the arguments, so any number of blanks may intervene between arguments. Also, the EXECUTIVE "folds" all lower case letters to upper case. However, if an argument is enclosed in quotes on the command line, then the entire quoted string is passed as a single argument, including blanks (if any), without folding alphabetic characters. The quotes themselves are stripped off. For example, if the command line was:

```
FIND "Now is the time for all good men"
```

```
then:    NCARG     will be 1
         CARG[1]   will be "Now is the time for all good men"
```

Command line arguments from the EXECUTIVE make a very useful way to pass file names or numeric values to a program. For example, here is a program segment which opens an input file specified on the command line for reading:

```
PROGRAM PROCESS
;   Program  segment to open a file passed as the first command line arg.

INCLUDE LIBRARY
WORD INFILE           ;Input file handle
BYTE LINE[81]         ;Buffer to hold a line from file
BEGIN
IF NCARG <> 1
  ABORT "#C***Error: PROCESS expects 1 argument which is a file name."
INFILE = OPEN(CARG[1])
IF INFILE = 0         ;open error?
  PUT NL, "*** Error: "
  CHOOSE IOERROR      ;error code from OPEN
  2
    PUT CARG[1]," is not a legal file name."
  3
    PUT "Disk drive not ready."
  4
    PUT CARG[1], " file not found."
  ELSE                ;unusual error of some kind
    PUT "Can´t open ",CARG[1]
  ABORT
WHILE GETLF (INFILE,LINE)  ; read lines until end of file
  ...
```

Of course, you could make the error processing simpler if you wished, or make it more sophisticated (perhaps by giving the user a chance to try another file name), as is appropriate to the application.

EXTERNAL VARIABLES FOR ADDRESSING SPECIAL MEMORY LOCATIONS

In BASIC you use PEEK and POKE to examine or set special memory locations in your computer. With PROMAL, you can write the equivalent of PEEK and POKE as follows:

BASIC	PROMAL
X=PEEK(nnnn)	X=M[nnnn]
POKE nnnn,X	M[nnnn]=X

where nnnn is the address of interest. The array **M** is predefined in the LIBRARY to be an array of bytes encompassing all memory, so M[0] is the first byte of memory, and M[65535] is the last byte of memory.

However, there is an even better way to replace those PEEKS and POKES which is both more readable and more efficient. You can give those special memory locations a variable name of type BYTE, by declaring them to be **EXTernal** to your program. For example for the Apple II:

```
EXT BYTE HIRESON AT $C057
```

defines a variable named HIRESON of type BYTE which will be **assigned** the address $C057. This is the Apple "soft switch" for enabling graphics mode. Once defined, you can enable hi-res mode by merely saying,

```
HIRESON=TRUE
```

which conveys a lot more meaning than POKE -16297,1.

INTERFACING TO COMMODORE 64 SPECIAL MEMORY LOCATIONS

PROMAL is very well suited for taking advantage of the special hardware features of the Commodore 64, such as sprites, music synthesis, and color. It is far easier to program these fun-filled features with PROMAL than BASIC. In BASIC, you depended on a lot of incomprehensible PEEKS and POKES to access the special registers in the VIC-2 video chip and the SID sound synthesizer. With PROMAL, you can give these registers a variable name and manipulate them just like any other variable. This kind of variable is called an EXTERNAL variable, because it is located outside the PROMAL program.

For example, the BASIC statement,

```
POKE 53281,7
```

sets the screen background color to yellow. With PROMAL, you might choose to do the equivalent function like this:

```
CON YELLOW = 7
EXT BYTE BACKGROUND AT 53281  ;Screen Background color reg.
...
BACKGROUND = YELLOW
```

Once you have defined the address of the variable BACKGROUND, you can use it just like any other PROMAL variable.

Creating animation with sprites is much easier with PROMAL. For example, suppose you wanted to have a tank moving horizontally on the screen as one sprite and a bomb falling vertically as a second sprite. You might do this as follows:

```
EXT BYTE XCAR AT $D000    ;X position of sprite 0
EXT BYTE YCAR AT $D001    ;Y position of sprite 0
EXT BYTE XBOMB AT $D002   ;X position of bomb
EXT BYTE YBOMB AT $D003   ;Y position of bomb
...
XCAR = XCAR + CARSPPEED   ;move car to right
YBOMB = YBOMB + BOMBSPEED ;move bomb down (+ is down)
...
```

In this case the address of each external variable was specified in hexadecimal, which is frequently more convenient.

You can also directly manipulate screen memory or color memory as a PROMAL array. For example, suppose you wanted to clear the standard screen, and then "paint" 16 bars across the screen, each in a different color:

```
PROGRAM RAINBOW
INCLUDE LIBRARY
CON SCREENSIZE = 1000          ; # of bytes in screen memory
EXT BYTE SCREEN [] AT $0400    ; C-64 screen memory location
EXT BYTE COLOR [] AT $D800     ; Color RAM
WORD I
BEGIN
FILL SCREEN, SCREENSIZE, ' '   ; fill the screen with blanks
FOR I = 0 TO 15
  FILL SCREEN+40*I, 40, $A0    ; one line of reverse-video blanks
  FILL COLOR+40*I, 40, I       ; set corresponding color for line
END
```

Notice that an external array declaration does not specify the size of the array inside the brackets. This is because PROMAL does not need to reserve any space within the program for this array (and because PROMAL does not do any bounds-checking on array references because this would adversely affect execution speed). The procedure FILL is a built-in LIBRARY subroutine which fills a portion of memory with a specified byte. Its first argument is the starting address, the second is the number of bytes to fill, and the last argument is the fill character.

Just for fun, let's compare the above program segment to its equivalent BASIC program:

```
 90 SC=1024: SZ=1000: CO=55296
100 FOR I=SC TO SC+SZ: POKE I,32: NEXT
110 FOR I=0 TO 15
120 FOR J=40*I TO 40*I+39: POKE SC+J,160: NEXT
130 FOR J=40*I TO 40*I+39: POKE CO+J,I: NEXT
140 NEXT I
```

If you run the BASIC program and the PROMAL program above, and time how long each takes to clear and paint the screen, you will find:

BASIC....about 14 seconds.
PROMAL...about 0.1 seconds.

This is another reason why PROMAL is much better than BASIC for animated graphics. PROMAL is much faster. While not every PROMAL program will be 140 times faster than its BASIC counterpart as in this example, speed increases of 20 to 100 times or more are commonplace. Also, the larger and more complex the program, the greater will be the relative speed improvement compared with BASIC. Using the built-in LIBRARY subroutines wherever possible will speed up your PROMAL programs even more, as well as making them smaller and easier to debug.

Several PROMAL demonstration programs making extensive use of the graphics and sound capabilities of the Commodore 64 can be found on the Commodore disk. You can learn a lot about PROMAL from studying these samples and improving them or changing them to suit your own taste.

PROMAL INTERFACE TO MACHINE LANGUAGE

Many BASIC programs have to resort to calling machine language subroutines for some specialized functions. Usually you need machine language routines because (1) BASIC is too slow, or (2) BASIC can't do what you wanted. Because PROMAL is so much faster than BASIC, and because it provides bit-level operators, BYTE data types, and EXTernal variables, you may never need any machine language at all with PROMAL.

If you **do** decide you need to call a machine language routine, PROMAL makes it much easier to do than BASIC. PROMAL provides a clean interface to machine language, both for routines you write and for ROM resident routines in your computers operating system. You can call any machine language routine in ROM without writing any machine language interface code at all. You can call machine language routines by name, with passed arguments, just like regular PROMAL routines. You can even specify the contents of the hardware registers and test the results when the machine language routines return (including the flags). You can embed machine language routines inside PROMAL programs using DATA statements or load them from separate files under program control. You can also call all the built-in PROMAL LIBRARY routines from your machine language routines.

Appendix I describes how to use and write machine language routines, with examples.

CHAPTER 7: STRINGS AND ARRAYS REVISITED

Earlier you saw how to declare and use strings and arrays. This section provides additional, more detailed information on using strings and arrays, especially multi-dimensional arrays and arrays of strings.

STRINGS

PROMAL always stores a character string as an array of bytes, one character per byte, plus a zero byte terminator indicating the end of the string. Strings are usually manipulated by specifying the address of the first character of the string. This is very convenient, since referring to an array name automatically generates a reference to the address of the first element. For example:

```
WORD I
BYTE BUF[81]  ; Input line string
...
GETL BUF      ; Input line from keyboard as string
I=BUF         ; I points to string
```

Assuming that this program was executed and that the user entered "Hello" from the keyboard, the memory for the variables might look like this, assuming some arbitrary addresses for the variables (shown in hex):

```
5B04    | 5B44 |   Variable I points to BUF

                                      5B94
5B44    | 48 | 65 | 6C | 6C | 6F | 00 | xx // xx |   Variable BUF = "Hello"
```

The "xx" in the diagram above means "don't care" or "undefined".

A big advantage of this representation of a string is that you can use the same array to either refer to the whole string, or to access single characters from the string. For example:

```
PUT BUF
```

will display "Hello", because the PUT procedure is passed the address of the string ($5B44 in the diagram above). Anytime you write the name of an array without any subscripts, the compiler uses the starting address of the array. You could extract a single character from BUF because it is an array of bytes. For example:

```
PUT BUF[1]
```

will display the character "e", the second character of the array (the first character is in BUF[0]). Alternatively, you could write **PUT (BUF+1)@<**, which would give the same result, because the expression will extract the value of the byte at $5B45 and print it. On the other hand, PUT BUF+1 would display "ello", because the value $5B45 would be passed to PUT instead of the contents of $5B45.

Remember that you can't use an assignment statement to copy a string from one variable to another (you need to use the MOVSTR procedure), but you can assign the address of a string to a word variable using an assignment statement. Therefore the the statements:

```
WORD I
BYTE BUF[81]
...
GETL BUF
I=BUF
PUT I
```

would cause whatever line was typed to be printed out. However, if these statements were followed by:

```
MOVSTR "Gone.", BUF
PUT I
```

then "Gone." would be printed, because I contains the address of BUF.

Similarly, you can't compare two strings with the ordinary comparison operators. For example:

```
DATA WORD BUF1 = "Hello"
DATA WORD BUF2 = "Hello"
...
IF BUF1 = BUF2  ; Wrong!
  PUT "Strings are the same"
```

will never print anything because the string given by BUF1 will never have the same address as the string given by BUF2, even though the contents of the strings are the same. This should be written as:

```
IF CMPSTR (BUF1, "=", BUF2)
  PUT "Strings are the same"
```

ADDRESS OF AN ARRAY ELEMENT VERSUS CONTENT OF AN ARRAY ELEMENT

It is very important to understand the difference between the address of an array and the value of an element in an array. Remember that almost all the built-in Library functions and procedures for string handling expect the **address** of the string for an argument (if the description of the routine says it expects a string, this means the **address** of the string). If you pass a single character where a string address is expected, you might create big problems! Here's an example:

```
BYTE BUF[81]
...
GETL BUF[0]    ; This is wrong!
...
PUT NL, BUF
```

Here the programmer simply intended to read in a line from the keyboard into the array BUF starting at the first element. The program compiles and appears to work, but always prints out garbage. In fact, sometimes it crashes the computer. Why?

The problem is that GETL expects the **address** of the buffer to receive the input, but the expression **BUF[0]** evaluates as the **value** of the first character of the buffer. In other words, whatever character is in BUF[0] when the call is made (some garbage value between 0 and 255) is passed to the GETL routine as the **address** of where you want the input line to go. GETL obliges by putting the input line someplace in the first 256 bytes of memory - but **not** in the BUF array. When BUF is printed by the PUT statement, it shows garbage, because it had never been set! Because the first 256 bytes of memory is "zero page" and holds critical operating system and PROMAL information, overwriting it may cause the computer to crash, necessitating a re-boot.

How do you fix this? In this case, the easiest way is simply to write:

```
GETL BUF
```

without the subscript, since PROMAL will always use the address of the array if you write its name without subscripts.

But what if you don't want the line to go right at the beginning of the array? Suppose, for example, you have a two dimensional array such as:

```
BYTE SCRN [81, 25] ; 25 lines of 80 characters each
```

Now suppose you want to input a line from the keyboard into the third row of the array. Here is how you do this:

```
GETL #SCRN[0, 2]  ; Read line from keyboard to 3rd row
```

The # operator tells the compiler to generate the **address** of the specified array element rather than the value of that element. The third element subscript is 2 instead of 3 because the first element is always 0, not 1. More importantly, remember that the last element is 24, not 25. If you try to read in the 25th line using:

```
GETL #SCRN[0, 25], 80  ; Wrong! Out of bounds!
```

GETL will oblige you by reading the line into memory over whatever happens to be in memory after the end of the SCRN array! This will have unpredictable and invariably unpleasant results.

To summarize, if a PROMAL Library routine (or any PROMAL subroutine for that matter) expects a string, you need to specify an address. If in doubt about how to make an address, place the # operator in front of the variable. If **X** is an array, then the following three statements are exactly equivalent:

```
PUT X
PUT #X
PUT #X[0]
```

All three statements will print the string which starts at the location of the X array (and is terminated by a zero byte). The following statement is **not** equivalent:

```
PUT X[0]
```

This statement prints only the first character of the string, because the expression will evaluate to the value of the first byte of the array X (which is assumed to be an ASCII character code). PUT is one of the few routines that can accept a single character or a string. If the argument is less than 256, PUT assumes the argument is a single ASCII character and prints it. If the argument is greater than 256, PUT assumes the argument is the address of a string to be printed.

SEQUENCE OF MULTI-DIMENSIONAL ARRAY ELEMENTS IN MEMORY

When using multi-dimensional arrays, note that the array elements with the **first** subscript will be adjacent in memory, so you should have the column subscript first and the row subscript second. For example:

```
BYTE PAGE [9,25]  ; Room for 25 lines of 8 chars each
...
PUT PAGE [0,5]  ; display single character on 6th line, first col
PUT #PAGE [0,5] ; display entire string of 6th line
```

Subscripts for the page array are allocated like this in memory:

0,0	1,0	2,0	...	7,0	0,1	1,1	2,1	...	6,24	7,24

Another way to look at this is to say that an array declared as:

```
WORD STUFF [10, 50]
```

declares 50 groups of 10 words each, **not** 10 groups of 50 words each. This distinction becomes important if you use BLKMOV to move part of the array or FILL to clear part of the array.

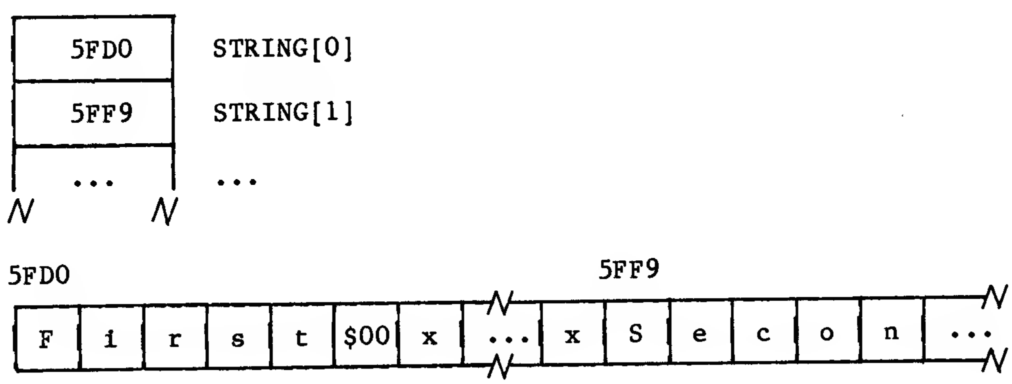
ARRAYS OF STRINGS

Sometimes, especially for sorting, you may wish to access an array of strings. Using an array of strings is usually more efficient than using a two dimensional array of bytes. The basic idea here is to use an array of type WORD which contains pointers into a singly-dimensioned array of type BYTE. This is especially true if you are simply going to sort the strings, because when a string is out of sequence, you can just exchange the pointers instead of exchanging entire strings. The sample program SORTSTRING.S provides a general purpose string sort routine which can sort an arbitrary array of strings passed as an argument.

Here is a program fragment showing how to develop an array of strings by reading them from the keyboard or a file. Input is terminated on end of file (CTRL-Z from the keyboard).

```
WORD STRING[100]  ; Array of strings (pointers into BUF array)
BYTE BUF[4100]    ; Storage for up to 100 strings of 40 char each
WORD I
WORD NSTRING      ; Actual number of strings (not exceeding 100)
...
NSTRING=0
I=BUF
WHILE GETLF(STDIN, I, 40) ; Read string to address I
  STRING[NSTRING]=I       ; Install pointer in string array
  I=I+41                  ; Starting addr of next string
  NSTRING=NSTRING+1
...
```

At the end of this program fragment, memory might look like this, assuming the lines read were "First", "Second", etc.



The SORTSTRING demo program uses a similar technique, but makes more efficient use of the BUF array. You may wish to use the sorting routine provided in SORTSTRING for your own programs.

PRESETTING GLOBAL VARIABLES TO ZERO

Unlike BASIC, PROMAL does not assign any initial value to variables declared in your program (except DATA, of course). Often you may wish to simply set all or a large number of variables to zero at the start of your program. Rather than writing assignment statments for each variable explicitly, here is a trick which will zero a block of variables. Assume you have a group of variables declared like this:

```
WORD FIRSTVAR
...
BYTE LASTVAR [1]
```

where FIRSTVAR is the first variable you want to zero and LASTVAR is a dummy variable you add after the last variable declared. At the start of your program, use:

```
FILL #FIRSTVAR, #LASTVAR-#FIRSTVAR, 0  ; Zero all variables
```

This will set all the variables from FIRSTVAR up to (but not including) LASTVAR to zero. Don't forget the # operators! Also note that LASTVAR is an array. This is necessary in the Commodore and Apple versions of PROMAL because the PROMAL compiler segregates scalar and array variables. It assigns addresses for all the scalar variables first (in the order declared), and then all the arrays (in the order declared). DATA variables are part of the code area of your program, not part of the data, so you don't have to worry about accidentally zeroing the value of any DATA identifiers.

CHAPTER 8: THE LOADER

INTRODUCTION TO THE PROMAL PROGRAM LOADER

PROMAL for Apple and Commodore 64 gives you, the programmer, the ability to control the loading and execution of programs. Your PROMAL program can load and run other PROMAL or machine language programs, or pieces of programs called **overlays**. Your program can also largely control **where** programs are loaded into memory, and specify what action should be taken when a program or overlay completes its task. Programs can call subroutines in other programs and use global variables in other programs previously loaded, and you can explicitly select which subroutines and variables can be used.

These capabilities are provided by the built in library procedure **LOAD**, which is used to control the loading, execution, and disposition of compiled programs and overlays. This procedure is extremely powerful, much more powerful than the simple "chaining" capability provided by BASIC and some other languages. To use it effectively requires an understanding of the loading and execution process as used by PROMAL, plus some new terminology. The remainder of this section deals with the LOADer. You may wish to skip over this section until you are familiar enough with PROMAL to be writing large programs.

DEFINITIONS

The following definitions are relevant to this section. The meaning of these terms will become clearer as the discussion develops.

A Module is the object file produced by the PROMAL COMPILER (with no error messages), with a .C extension, or a relocatable machine language program as generated by the RELOCATE program (discussed in **Appendix I**).

An Entry Point is the place where execution begins in a module. In a PROMAL source program, the entry point is represented by the BEGIN statement following the last procedure or function in the source program.

A Logical Program is a collection of one or more modules which, taken together, comprise a logically complete program for some purpose. A logical program may have several modules, each residing on disk in a separate file. As a minimum, a logical program has one module.

A Program is a compiled PROMAL program. Normally it performs a complete task by itself and is composed of an arbitrary number of procedures and functions, with exactly one entry point, which is at the BEGIN statement following the last procedure or function. The program source file begins with a PROGRAM statement, is compiled from one or more source files, and the resulting output module is contained in a single object file with a .C extension.

An Overlay is a piece of a complex logical program which is kept on disk until it is needed, and is then loaded into memory and executed under program control. The overlay source file begins with an OVERLAY statement, and is otherwise similar to a program. It has an arbitrary number of procedures and functions and exactly one entry point, which is at the BEGIN statement following the last procedure or function. It must be compiled separately from the rest of the logical program which is associated with it, from one or more source files, and the resulting output module is contained in a single object file with a .C extension.

Loading is the process of taking a PROMAL module from disk and copying it into memory, making any adjustments (called relocations) to the program which are needed to correct addresses in the program or interface to other modules, and transferring control of execution to the program, if desired.

Chaining is a special kind of loading where the program being loaded replaces the program which called the loader.

BREAKING UP A LOGICAL PROGRAM INTO MODULES

There are several reasons why you might want to have a logical program composed of several modules instead of a single, monolithic program:

1. The program is too large to fit in memory all at once.

2. The program is composed of logically separate modules (for instance, an accounting system might have a main menu with separate modules for receivables, payables, order processing, report generation, etc).

3. The program uses a logically-related group of subroutines which are not frequently changed and therefore do not need to be re-compiled (for example, the PROMAL graphics package or real functions).

4. The program takes too long to compile in its entirety.

5. The program uses large machine language routines (this is discussed in Appendix I).

HOW THE PROMAL LOADER WORKS

The PROMAL LOADer is a built-in procedure in the Library, called the same way as other library routines. You have already seen the PROMAL loader working, at least indirectly. When you type the name of a program you want executed from the EXECUTIVE, the EXECUTIVE calls the LOAD procedure to run your program. When your program finishes, it returns through the LOADer to the EXECUTIVE at the point from which it was called.

Your programs can in turn load and run other programs or overlays, by specifying the name of the program to run, and optionally some flags indicating **how** the program should be run. Briefly, the LOADer performs these tasks:

1. Looks to see if the specified program or overlay is already in memory, and if so, executes it beginning at the entry point. Otherwise, it:

2. Locates the specified module on disk and determines how much memory is needed for your program and its variables. If there is not enough room, it unloads other programs (unless the module is an overlay) until there is enough room.

3. The memory image of the module is copied from disk into the available memory space.

4. The loader then reads tables which follow the memory image on disk to determine what adjustments are necessary to the memory image. These adjustments are called relocations, and are needed to install the correct addresses for branch instructions and subroutine calls. The loader can also adjust address references to other modules already loaded (using EXPORTS and IMPORTS, discussed later).

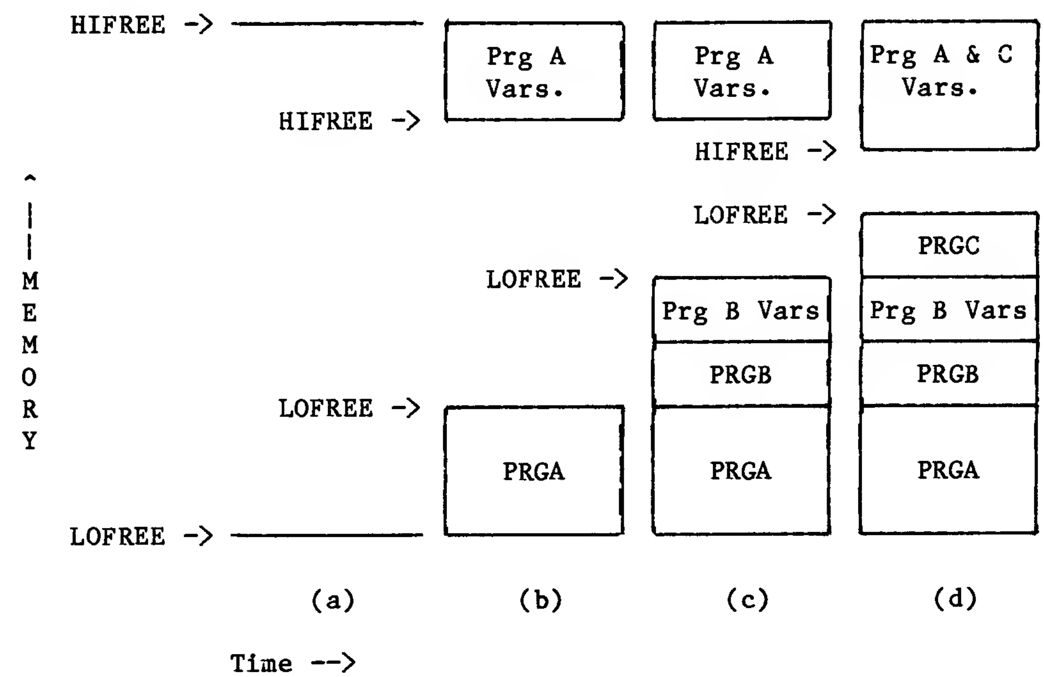
5. The loader then begins execution of your program at the entry point.

6. When your program completes (by coming to the END of the main program, calling EXIT or ABORT, or encountering a runtime error), the loader temporarily regains control. It normally transfers control back to the program which called the loader at the statement following the call to the loader. However, if your program called ABORT or encountered a runtime error, control is passed directly back to the EXECUTIVE instead.

The program which calls the LOADer to execute another program is called the **parent** of that program. The new program is called the **offspring** of the program that called the loader. The loader keeps track of the modules currently in memory by a series of tables. These tables have room enough for up to **six** modules to be resident in memory at once, plus the EXECUTIVE and EDITOR. These modules may be all part of one logical program, completely separate programs, or any combination.

The LOADer also uses several pointers for memory management. The most important of theses are called **LOFREE** and **HIFREE.** LOFREE always points to the first byte of unused memory, and HIFREE always points to the byte **after** the last unused byte of memory. These pointers always point to a page boundary in memory (that is, the address is of the form $XX00). Normally the LOADer allocates programs from the bottom of available memory up, and variables from the top down. Normally variables from one module can occupy the same memory as variables for another program since the variables have no initial value and the programs are not related. However, the key word OWN on the PROGRAM or an OVERLAY declaration of a program can be used to force the loader to allocate the variables immediately after the program, not shared with any other programs. OVERLAYs always have their variables allocated immediately after the overlay code.

The following **memory diagram** shows a series of programs being loaded from the EXECUTIVE:



Time is represented on the horizontal axis and memory on the vertical axis, with the highest addresses at the top. The diagram represents the memory configuration for the Apple. The Commodore 64 configuration is somewhat more complicated (see **Appendix G**), because the Workspace is also managed by the LOADer, but the principle is the same. In this diagram, there are initially no programs in memory (except the EXECUTIVE/EDITOR, not shown). Then PRGA is executed (part b of the figure), which is a normal module. PRGB is then executed. PRGB has OWN on its program line, so its variables are allocated after the code for PRGB instead of sharing its variable space with PRGA. Finally, PRGC is run, which is another normal program, and shares its variable space with PRGA. Since PRGC requires more variable space than PRGA, HIFREE is lowered by the loader. HIFREE will always point to the start of the variables for whatever program requires the largest block of shared variable space.

In this example, all the programs were small enough to fit in memory at once. If PRGC had been too large to fit, the loader would first unload PRGB and its variables and try again. If there was still not enough room, it would unload PRGA. When a program is unloaded, the LOADer simply deletes its table entry and moves the LOFREE pointer down (and the HIFREE pointer up, if possible), to recover the space. It does not clear the memory recovered.

HOW TO CALL THE LOADER

The declarations needed to access the LOADer are in a file called PROSYS.S on the PROMAL system disk. Therefore you should have:

```
INCLUDE PROSYS
```

near the top of any program which will be calling the loader (or, if you wish, you can extract the definitions from PROSYS and insert them directly into your program with the EDITor).

The loader is a built-in procedure in the PROMAL library, which you call with a statement of the following form:

LOAD Progname [,Bitflags]

where **Progname** is a string containing the desired module name (without the file extension), and **Bitflags** is an optional argument of type BYTE which contains several flags, described shortly. If the Bitflags argument is not specified, it defaults to 0, for a normal load-execute-return sequence. The loader also sets a special variable called LDERR (also defined in PROSYS), as follows:

LDERR #	Meaning
0	No error, module was successfully loaded/executed.
1	Module was not found in memory or on disk (or the name is illegal)
2	Not a valid PROMAL module (e.g., not a successfully compiled program).
3	Not enough free memory to load program.
4	Module required not loaded or relocation error (e.g., the module to be loaded calls a subroutine in another module which is not loaded).

For example,

```
PROGRAM MYPROG
INCLUDE LIBRARY
INCLUDE PROSYS
...
BEGIN
...
LOAD "YOURPROG"
IF LDERR <> 0
  ABORT "#C Unable to load YOURPROG"
...
END
```

will cause the module YOURPROG.C to be loaded into memory (if it is not already there) and executed. After YOURPROG ends, control will return to the IF statement following the call, which tests for a loader error (such as file not found). If, however, YOURPROG called ABORT or encountered a runtime error, control would never return to the IF statement above, but would return directly to the EXECUTIVE instead.

LOADER OPTIONS USING BIT FLAGS

The second, optional argument of type BYTE can be used to specify a variety of options which control the loading process. This byte is treated by the LOADer as several one-bit TRUE/FALSE flags. These flags are given names in PROSYS, defined as follows:

Name	Definition	Meaning to LOADer
LDCHAIN	$01	Chain to program. If TRUE (1), the calling module should be unloaded and replaced with the new module. When the new module ends, control should return to the **parent** of the calling module.
LDPRCLR	$02	Pre-clear memory. If TRUE (1), all programs in memory (including the caller) should be unloaded before loading the specified program. Control will be returned to the EXECUTIVE. This option is usually used to guarantee the maximum available memory for a program.
LDRELD	$04	Re-load module. If TRUE (1), the specified module should be reloaded from disk, even if it already is in memory. If FALSE (0), it will not be reloaded from disk unless it is not already loaded or the memory-resident copy has been corrupted. Note that specifying LDCHAIN=1 or LDPRCLR=1 also implies LDRELD=1 automatically.
LDRECLM	$08	Reclaim memory on exit. If TRUE (1), then the specified module should be unloaded from memory after it completes execution. This option is normally used for overlays to make room for other overlays in the same memory space. If 0, the module will remain loaded on completion and can be re-executed by a subsequent LOAD without having to access the disk.
LDNOGO	$10	Do not execute. If TRUE (1), then the specified module will be loaded into memory (if it is not already loaded) without executing it. This option is normally used to insure that a module is loaded and ready for later execution. It is also used to control the sequence of loading of multiple modules in a complex logical program. If 0, the specified module will be executed.
LDUNLD	$20	Unload. If TRUE(1), then the specified module is unloaded instead of loaded. No other action takes place and all other bit flags are ignored. Note that any program loaded above the specified module will also be unloaded. If the calling module is itself unloaded as a result of this process, control will return to the parent program instead. This option is normally used to free up additional memory.

The bit flags above can be combined in any sensible combination. For example:

```
LOAD "HISPROG", LDCHAIN + LDRECLM
```

will remove the calling program from memory, load and execute HISPROG, then remove HISPROG from memory and return control to the parent of the original caller (probably the EXECUTIVE).

```
LOAD "NEXTCOMD", LDPRCLR+LDNOGO
```

will unload all programs from memory, load NEXTCOMD without executing it, and return control to the EXECUTIVE (you might want to do this to setup the next program to be run in memory).

USING VARIABLES, PROCEDURES AND FUNCTIONS IN OTHER MODULES

One of the most powerful features of the PROMAL LOADer is that when a module is loaded and executed, it can call selected procedures and functions and access selected variables in other modules which are already loaded. It cannot, however, reference procedures, functions, or variables in modules which have not been loaded yet. This is a logical extension of the rule that functions, procedures and variables must be defined before they are referenced. It is up to you, the programmer, to determine which procedures, functions, and variables will be made available to other modules. This is done using EXPORTs and IMPORTs.

EXPORTS AND IMPORTS

In a PROMAL source program, the key word **EXPORT** can be used in front of any declaration of a constant, data declaration, variable, procedure, or function to designate an item which should be made available to other modules which wish to use it. If your program contains **any** EXPORTs, it **must** also have the keyword EXPORT on the PROGRAM (or OVERLAY) line. OWN should also be specified.

For purposes of illustration, let us assume we will have a logical program composed of two separate modules. The first module is a collection of subroutines which you frequently use, called SUBPKG, and the other module is a particular application program called MYPROG, which will use some routines in SUBPKG (and in addition has some procedures, functions and global variables of its own). Assume you wish to compile the subroutine package and MYPROG separately, because SUBPKG is already well-debugged and is fairly large. Therefore during development of MYPROG you will not have to re-compile SUBPKG each time you make a change to MYPROG, saving time. Here is a skeletal view of the source for SUBPKG:

```
PROGRAM SUBPKG OWN EXPORT
INCLUDE LIBRARY
...
WORD I
EXPORT WORD CLEARANCE
EXPORT CON WORD POOLSIZE=500
EXPORT REAL POOL[POOLSIZE]
REAL THRESHOLD
EXPORT DATA REAL PI = 3.1415926535
EXPORT DATA WORD ERRMSGS [] =
"Pool exhausted", "Undefined pool element", "Illegal pool element",0
...
```

```
...
EXPORT PROC ADDTOPOOL ; Item
ARG WORD ITEM
...
END
...
FUNC REAL BESTGUESS
...
END
...
EXPORT FUNC BYTE CHECKERROR
...
END
...
BEGIN
...
END
```

This example illustrates how a subroutine package might make selected identifiers available for use by other, separately compiled modules. In this case, the names CLEARANCE, POOLSIZE, POOL, PI, ERRMSGS, ADDTOPOOL, and CHECKERROR will be exported. The names I, THRESHOLD, and BESTGUESS will not be available for use by other modules. In other words, the subroutine BESTGUESS can be called by other routines in this module (PROGRAM SUBPKG), but not by other separately compiled modules.

When a program contains EXPORTs, the PROMAL COMPILER writes the definitions of **all** the exported items to a special text file at the completion of compilation. This text file will have the same name as is on the PROGRAM declaration in the source file, but with a .E extension. For example, the program above would cause the compiler to generate an export file called **SUBPKG.E**, which might look like this:

```
IMPORT SUBPKG ;10/17/85
  EXT FUNC BYTE CHECKERROR AT $0562
  EXT PROC ADDTOPOOL AT $0250
  EXT DATA WORD ERRMSGS [4] AT $000D
  EXT DATA REAL PI AT $0007
  EXT REAL POOL [500] AT $0000+$4
  CON WORD POOLSIZE = $01F4
  EXT WORD CLEARANCE AT $0002
```

This file tells the definitions of the exported identifiers, relative to the start of the SUBPKG module. It is not necessary to understand the exact meaning of the individual lines. The top line tells the name of the exporting program and the compilation date.

IMPORTING DEFINITIONS

Once the export file has been written by the COMPILER, you may INCLUDE it in the compilation of another, separate module, to import all the desired definitions. For example, another separately-compiled program which uses the SUBPKG module might look like this:

```
PROGRAM MYPROG
INCLUDE LIBRARY
...
INCLUDE SUBPKG.E
...
WORD K
...
PROC AJUSTPOOL
BEGIN
...
POOL[K] = PI/4.
ADDTOPOOL
...
END
...
BEGIN
...
IF K > POOLSIZE
  PUT NL,ERRMSGS[0]
...
END
```

Notice that this program uses the procedure ADDTOPOOL, the data items PI and ERRMSGS, and the constant POOLSIZE without ever explicitly declaring them. This is possible because the INCLUDE SUBPKG.E will cause the definitions to be imported. Please note that you must specify the .E extension on the INCLUDE statement; otherwise, the compiler will look for the file SUBPKG.S instead by default.

EXECUTING THE LOGICAL PROGRAM WITH SEPARATE MODULES

After compiling MYPROG, you will have two separate modules which work together: SUBPKG.C and MYPROG.C. If you attempt to execute MYPROG.C from the EXECUTIVE, you will get the message:

```
NOT LOADED OR RELOC ERROR: SUBPKG
```

This is because the SUBPKG module must be loaded before the MYPROG module which calls it, and you haven't loaded it. To solve this problem, you could type:

```
UNLOAD
GET SUBPKG
MYPROG
```

which would unload any existing programs (to make sure there's enough room for both modules), load the SUBPKG module, and then load and execute the MYPROG module. The LOADer is able to relocate all the references to routines in SUBPKG correctly because (1) it knows where it loaded SUBPKG into memory, and (2) it knows the definitions of the references to the exported items in SUBPKG as a result of the compilation of MYPROG.

In the example above, it is important to understand that the exported routines in SUBPKG can be called from the MYPROG module, but that no routines in MYPROG may be called from SUBPKG, even if you EXPORT them. This is because the module doing the exporting must always be loaded before the module doing the importing, and it is clearly impossible for both modules to be loaded first!

USING A BOOTSTRAP TO CONTROL LOADING

If you have an application program with several modules which need to be loaded in a certain order, you may want to write a **bootstrap** program, whose job is to load all the needed modules in the proper sequence and then run the main program. Typically a bootstrap program might do this:

1. Display a signon message and any information (such as a menu of choices) relevant to the program, that the user can read while the rest of the program is loading from disk.

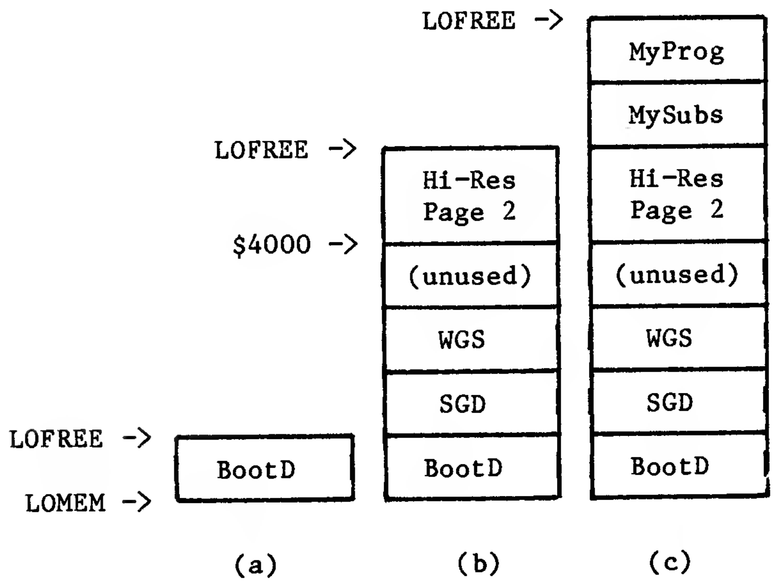
2. Load the modules needed in the desired order, using the LDNOGO option on each LOAD call to prevent execution.

3. Load and execute the main module.

In some cases, you may even want to use a two-stage bootstrap loader, where the first stage bootstrap loader signs on and then LOADs the second bootstrap stage loader with the LDPRCLR option to insure all possible memory is available for the application. The second bootstrap then loads all the modules needed to get the program going, in the correct order to resolve all the dependencies.

In some cases you may also use the bootstrap loader to directly manipulate the LOFREE pointer before LOADing some modules to reserve certain areas of memory. For example, on the Apple II using graphics you may wish to load part of your logical program below the 8K graphics page from $4000 to $6000, and part above it. The following memory diagram, with program fragment to the right, illustrates this (and other examples may be found in the PROMAL GRAPHICS TOOLBOX Manual).

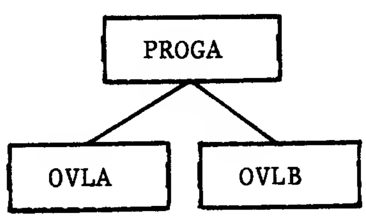
After programs BOOTD, MYSUBS, and MYPROG have been compiled (in that order), then executing BOOTD from the executive will cause BOOTD to load the SGD and WGS modules, as shown in (b) above, then sets LOFREE to $6000 under program control to reserve the Apple Hi-Res screen area, and finally loads the MYSUBS and MYPROG module to begins execution of MYPROG. Naturally you should adjust the LOFREE pointer only with a good understanding of what you are doing!




```
PROGRAM BootD
INCLUDE PROSYS
...
LOAD "SGD",LDNOGO
LOAD "WGS",LDNOGO
LOFREE=$6000 ;end hires
LOAD "MySubs",LDNOGO
LOAD "MyProg"
END

PROGRAM MySubs OWN EXPORT
...
INCLUDE SGD.E
INCLUDE WGS.E
EXPORT PROC MyProc
...
END

PROGRAM MyProg
...
INCLUDE SGD.E
INCLUDE WGS.E
INCLUDE MySubs.E
...
MYPROC
...
END
```

USING OVERLAYS

In the previous example, separate compilation was used primarily as a convenience. Sometimes, it is a necessity. This usually happens when a logical program is simply too large and complex to fit into the available memory space all at once. When this happens, the usual solution is to use overlays. A logical program which uses overlays will have one or more modules which remain resident in memory throughout execution, and will switch other modules in and out of memory as needed. A typical overlaid program might be organized like this:



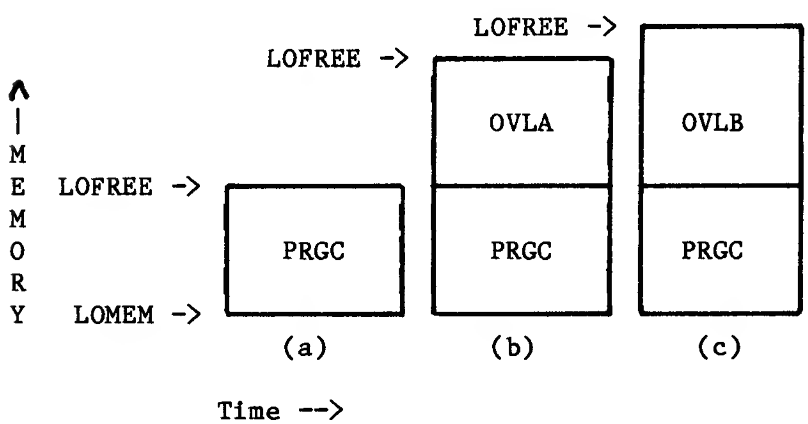
This tree diagram indicates that the logical program has a **root module**, PROGA, and two mutually-exclusive overlays, OVLA, and OVLB. By mutually exclusive, we mean that only one of the overlays will be in memory at any given time. Therefore the overlays can all share the same memory space, so that the total space needed will only be equal to the size of the largest overlay, instead of the sum of the overlays.

As far as the source code for an overlay is concerned, the only difference between a program and an overlay is that the first line of the program should contain the key word OVERLAY instead of PROGRAM, for example:

```
OVERLAY OVLA
```

The OVERLAY keyword also has the effect of including the OWN keyword on the program declaration line. Otherwise, the loader would allocate the variables belonging to the overlay right on top of the variables used by the rest of the logical program, probably producing a disaster.

As far as the loader is concerned, there is only one difference between a PROGRAM and an OVERLAY. The LOADer will **not** automatically unload any modules to make room for an overlay. If it did, there would always be the possibility that the loader would have to unload the calling module to make room for the overlay. This is contradictory to the normal use of overlays, which normally return to the parent module when completed.

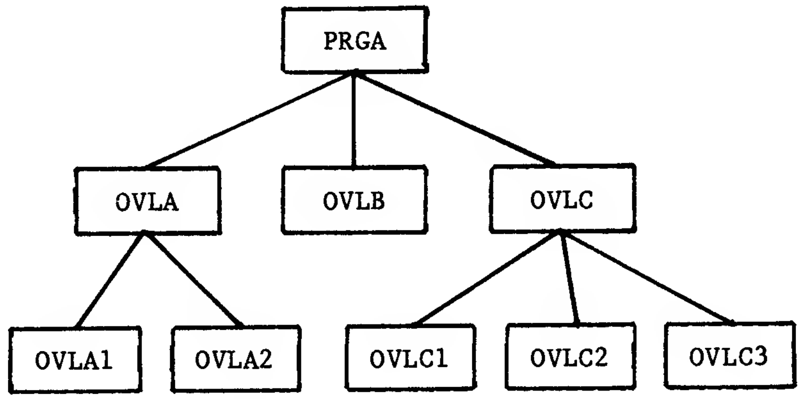



```
PROGRAM PRGC OWN EXPORT
INCLUDE PROSYS
...
LOAD "OVLA", LDRECLM
...
LOAD "OVLB"
...
END

OVERLAY OVLA OWN
INCLUDE PRGC.E
...
END

PROGRAM OVLB OWN
INCLUDE PRGC.E
...
END
```

In our example program with 2 overlays, the memory diagram might look as shown above, with the program skeleton to the right.

Note that the LDRECLM option was specified on the call to the LOADer. This is the normal way to load an overlay which will be replaced by another overlay later. Remember that the LOADer will not unload anything (including another overlay) to make room for an overlay; therefore you will probably want to specify LDRECLM to insure that the overlay is unloaded when it is completed. Of course, there is always the possibility that the root module might want to call the same overlay again. In this case, you might want to consider leaving the overlay in memory when it completes. If you need it again, the LOADer won't have to actually load it. If you need to replace it with a different overlay instead, you can unload it explicitly using the LDUNLD option, before loading the desired overlay.

The sample program fragment above had only two overlays. In a complex application, there might be several overlays, or even two layers of overlays, as shown by the tree below:



In this case, the overlays OVLA, OVLB and OVLC might export variables and subroutines to the five overlays at the bottom of the diagram. In this case, you would need to have the keyword EXPORT on the OVERLAY declaration:

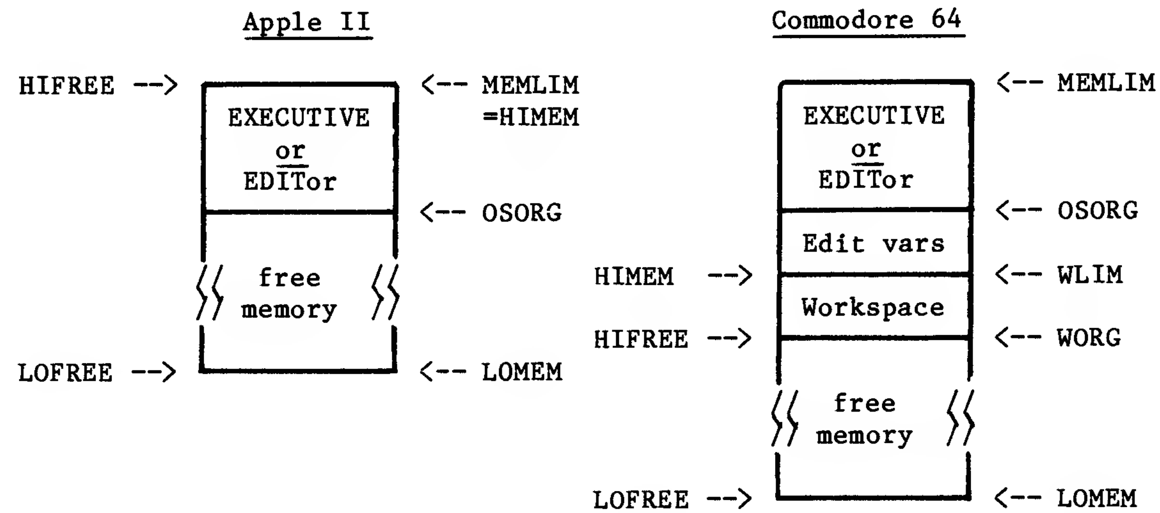
```
OVERLAY OVLA EXPORT
```

CONSIDERATIONS FOR THE EXECUTIVE AND EDITOR

This section tells you how it is possible to load larger programs under program control than it is possible to LOAD using the EXECUTIVE GET command, by overwriting the space usually reserved for the EDITor. The system pointers referred to below are defined in PROSYS.S. Further memory map information is included in Appendix G.

As was indicated before, the LOADer always considers the free, allocatable space to be between LOFREE and HIFREE. The EXECUTIVE and EDITor **both occupy the same address space**, which is OSORG through MEMLIM (about 11.5K bytes), but only one or the other of these programs occupies this space at any one time.

For the Apple II, a copy of the EDITor and the EXECUTIVE is kept in the extra 64K memory bank, and each is copied back into the normal address space at OSORG when needed (this does not apply for applications programs generated using the GENMASTER program in the Developer's package). Since the EXECUTIVE or EDITor are always copied into main memory when needed, all the space, including that used by the EXECUTIVE or EDITor, is available for allocation for your programs by the LOADer (about 25K). If you use the GET command to load a program which overlaps OSORG, however, it will be immediately destroyed when the EXECUTIVE is copied back into memory when the GET command is completed. The Workspace for the Apple II is kept in the extra 64K bank, so it is of no concern.



Memory with no programs loaded

For the Commodore 64, the situation is somewhat more complicated. The Editor is kept in the 12K byte section of RAM under the ROMs from $D000 to $FFFF when the EXECUTIVE is active. When any program is executed, the EXECUTIVE is swapped with the EDITOR. The EXECUTIVE is therefore "hidden" under the ROMs when your programs run, and the EDITOR occupies the space from OSORG to MEMLIM. When control returns to the EXECUTIVE, it is swapped with the EDITor again. For the Commodore, the Workspace is also allocated near the top of "free memory".

For large logical programs on the **Commodore 64**, you may want to let the LOADer use the space normally set aside for the EDITOR for your program(s). This can be done by using a bootstrap program (as described above), which should be the only program loaded (you can use a two-stage bootstrap to guarantee this). This bootstrap program might have the following form:

```
PROGRAM BOOTBIG OWN
INCLUDE LIBRARY
INCLUDE PROSYS
...
WPTR=WORG
WEOF=WORG       ; Make workspace empty
WSIZE=0         ; No workspace usable
EDRES=FALSE     ; EDITor will no longer be ther
HIFREE=MEMLIM   ; Reclaim Editor's space
...
```

It would then Load and execute your programs, described in the section on bootstrap programs, above.

When your program returns control to the EXECUTIVE, the EXECUTIVE will move back into memory at OSORG. If you subsequently use the EDITor, it will automatically be re-loaded (from disk, for the Commodore 64).

A GENERAL PURPOSE COMMODORE BOOTSTRAP FOR BIG PROGRAMS

Here is a bootstrap program which can be used to load and run a program which is too large to run on the Commodore 64 without overwriting the EDITOR. The program to be run **must** have OWN on the PROGRAM line. The Workspace will be cleared and the EDITor overwritten. To use the bootstrap, first give an **UNLOAD** command, and then type:

```
BOOTBIG Progname
```

from the EXECUTIVE, where Progname is your large, compiled program.

```
PROGRAM BOOTBIG OWN  ;Commodore 64 only boot big program
; Kills workspace and EDITor
INCLUDE LIBRARY
INCLUDE PROSYS
BEGIN
IF NCARG <> 1
  PUT NL,"BOOTBIG ABORTED: No name given"
  ABORT "#CUsage: BOOTBIG Progname"
HIFREE=MEMLIM  ;The max memory please
WORG=MEMLIM    ;No workspace
WPTR=MEMLIM
WEOF=MEMLIM
WLIM=MEMLIM
WSIZE=0
EDRES=FALSE    ;EDITor not resident
LOAD CARG[1]   ;Load & execute
IF LDERR <> 0
  ABORT "#cBOOTBIG LOAD ERROR $#H",LDERR
END
```

REMINDERS FOR SUCCESSFUL USE OF OVERLAYS AND SEPARATE COMPILATION

1. The root module or modules need to export any definitions needed by the overlays, and the overlays each need to INCLUDE the exports (don`t forget the .E extension).

2. Each overlay must start with:

```
OVERLAY Name [EXPORT]
```

and must be compiled separately.

3. Overlays may call routines and use variables exported from the root module(s), or other overlays already loaded. The root module cannot contain calls to routines or reference variables declared in the overlays. The **only** way to enter an overlay is by a call to LOAD, which will transfer control to the entry point of the overlay.

4. Remember that **if you alter one module, no matter how trivial the change, you must re-compile all modules which access it**. This is entirely the programmer's responsibility; there is no way for PROMAL to check it for you. If you fail to do this, the LOADer will relocate the program incorrectly, probably resulting in mysterious crashes when your program runs. One way to

check for this is to look at the date which the compiler writes on the first line of the export file (.E extension). If this date is later than the compilation date on any of the modules which INCLUDE it, you need to recompile those modules.

5. If you manipulate LOFREE or HIFREE, remember that the low order byte must always be 0 (i.e., always points to a page boundary).

6. You should always check the value of LDERR after any call to LOAD, and print an appropriate diagnostic message if an error occurs.

7. Be sure to INCLUDE PROSYS (or copy the definitions from PROSYS.S directly into your source program) for any program using the loader.

8. The LOAD procedure depends on the underlying operating system, memory map and computer hardware for its operation. Therefore you should not expect programs using LOAD or EXPORT to necessarily be completely portable to other kinds of computers or operating systems, just because PROMAL is available on that computer.

9. If you EXPORT anything, you must have EXPORT on the PROGRAM (or OVERLAY) line, or you will get an "ILLEGAL EXPORT" error when the COMPILER encounters the first EXPORT declaration. A PROGRAM declaration should also have OWN specified (or else the variables will be assigned the same addresses as any other module not specifying OWN).

10. Exporting scalar variables may increase the memory usage of your program somewhat.

11. A maximum of six modules may be in memory at once (8 with a program generated with GENMASTER in the Developer's package).

12. For the Commodore 64, using LOAD with the LDPRCLR option, the LOADer will set HIFREE back to HIMEM after unloading all programs, to preserve the space normally occupied by the EDITor. If you want the EDITor space to be available for loading too, you need to set HIMEM to MEMLIM before calling the LOADer (Caution: this will cause the Workspace (if any) to be moved up to the top of memory too, and it will be clobbered when the EXECUTIVE swaps back in). Your program should restore HIMEM before exiting back to the EXECUTIVE.

13. If you use multiple modules and have an ESCAPE in one module to a REFUGE in a separate module, if you exit from the module with the ESCAPE via a normal END, the program will still return to the parent program of the original module.

P R O M A L

(PROgrammer's Micro Application Language)

LIBRARY MANUAL

For Apple II and Commodore 64 Computers

SYSTEMS MANAGEMENT ASSOCIATES, INC.
3325 Executive Drive
Raleigh, North Carolina 27609
(919) 878-3600

Rev. C - Sep. 1986

PROMAL LIBRARY

The PROMAL system contains a LIBRARY of predefined procedures and functions which are automatically loaded into memory when PROMAL boots up. This library provides input, output and utility routines which greatly ease the programmer's job. To use any or all of the LIBRARY routines, you simply need to add the statement:

```
INCLUDE LIBRARY
```

near the start of your program. This statement tells the COMPILER to read the definitions for the standard library. This is just a list of external procedures and subroutines. You can display the list with a **TYPE L** command from the EXECUTIVE.. The INCLUDE LIBRARY statement does NOT make your program any larger. The library routines are always present in memory; the INCLUDE LIBRARY statement merely tells the compiler what the names are and where they are.

Once you have defined the routines in the LIBRARY with the INCLUDE statement, you may freely use them in your program. You call LIBRARY routines in the same manner as other PROMAL routines. Unlike normal PROMAL routines, however, most LIBRARY routines can be called with a **variable number of arguments**, some of which are optional. For example, the LIBRARY function PUT can have one or more arguments. In most cases, the optional arguments have a default value, which will suffice for most cases. For special cases, you can specify additional arguments which modify the way the routine works.

Table 1: LIBRARY Summary

Name	Avail.*		Description
ABORT	**AC**	L	Abort program execution, optionally displaying message.
ABS	**AC**	L	Absolute value of REAL value.
ALPHA	**AC**	L	Test if character is alphabetic.
BLKMOV	**AC**	L	Block move.
CHKSUM	**AC**	L	Compute 16 bit checksum of memory region.
CLOSE	**AC**	L	Close an open file or device.
CMPSTR	**AC**	L	Compare strings.
CURCOL	**AC**	L	Determine current cursor column.
CURLINE	**AC**	L	Determine current cursor line.
CURSET	**AC**	L	Position cursor on the display.
DIR	**AC**	L	Display file names matching a pattern
DIROPEN	**A**	P	Open disk directory for reading.
EDLINE	**AC**	L	Edit a line on the screen.
EXIT	**AC**	L	Exit from program, optionally with message.
FILL	**AC**	L	Fill a memory block with a constant.
FKEYGET	**AC**	L	Get a current function key definition.
FKEYSET	**AC**	L	Define a function key expansion string.
GETARGS	**AC**	L	Split command line into arguments.
GETBLKF	**AC**	L	Block-read from file.
GETC	**AC**	L	Input one character from keyboard with echo to screen.

GETCF	**AC**	L	Input one char. from a file or device.
GETKEY	**AC**	L	Input character from keyboard, no echo to screen.
GETL	**AC**	L	Input one line from the keyboard.
GETLF	**AC**	L	Input one line from a file or device.
GETPOSF	**A**	L	Obtain current file position.
GETTST	**AC**	P	Test if T device is ready (serial port)
GETVER	**AC**	P	Return PROMAL version and machine code.
INLINE	**AC**	L	Input a line with screen editing.
INLIST	AC	P	Search linked list.
INSET	**AC**	L	Test if a character is in a string.
INTSTR	**AC**	L	Convert signed value to a string.
JSR	**AC**	P	Call machine language subroutine.
LENSTR	**AC**	L	Return length of string.
LOAD	**AC**	P	Load/unload/execute program or overlay.
LOOKSTR	**AC**	L	Search a list of strings.
MAX	**AC**	L	Return the largest of two or more arguments.
MIN	**AC**	L	Return the smallest of two or more arguments.
MLGET	**AC**	P	Load machine language program or memory image.
MOVSTR	**AC**	L	String copy or substring or concatenate strings.
NUMERIC	**AC**	L	Test if character is numeric.
ONLINE	**A**	P	Get ProDOS volume name for specified disk drive.
OPEN	**AC**	L	Open a file or device for input/output.
OUTPUT	**AC**	L	Formatted output with many options.
OUTPUTF	**AC**	L	Formatted output to a file.
PROQUIT	**AC**	P	Exit from PROMAL system.
PUT	**AC**	L	Output text to the display.
PUTBLKF	**AC**	L	Block-write to file.
PUTF	**AC**	L	Output text to a file or device.
RANDOM	**AC**	L	Obtain a pseudo-random number.
REALSTR	**AC**	L	Convert a REAL value to a string.
REDIRECT	**AC**	P	Redirect standard input/output to file/device.
RENAME	**AC**	L	Rename a file.
SETPOSF	**A**	L	Set desired position in a file (random access).
SETPREFIX	**A**	L	Set the path name for directory searches.
STRREAL	**AC**	L	Convert a string to a REAL numeric value.
STRVAL	**AC**	L	Convert a string to a numeric value.
SUBSTR	**AC**	L	Search for one string in another.
TESTKEY	**AC**	L	Test if a key is pressed on keyboard.
TOUPPER	**AC**	L	Fold lowercase letter to upper case
WORDSTR	**AC**	L	Convert an unsigned value to a string.
ZAPFILE	**AC**	L	Delete a file.

------

* **Note:** Avail. meaning: First column indicates machine availability, A=Apple II, C=Commodore 64. Second column indicates the required INCLUDE file, L=LIBRARY, P=PROSYS.

A few unusual or system-dependent routines are defined in a separate file called PROSYS.S. These routines are also always resident in memory, but in order to use them you need to add the statement:

```
INCLUDE PROSYS
```

near the beginning of your program. This file also defines some less-frequently-needed system variables.

The LIBRARY routines are summarized above. The "Avail." column indicates which computers are supported (Apple or Commodore) and which INCLUDE file is needed in order to use the routine (LIBRARY or PROSYS).

HOW TO USE THE LIBRARY ROUTINE DESCRIPTIONS

The following section provides a detailed description of each routine in the LIBRARY, in alphabetical order. The USAGE line gives the syntax for calling the routine. Words shown in CAPITAL LETTERS are required to be entered as shown (although they do not have to be typed using upper case letters). Words shown in lower case with the first letter capitalized are user-supplied arguments. These arguments can be variables or constants or more complex expressions. Arguments shown in square brackets, [and], are optional arguments which may be included or left off at the programmer's discretion. The description of the routine will tell what actions are taken if the optional arguments are not supplied. Ellipsis (...) are used to indicate an arbitrary number of repetitions of an optional argument. For functions, the USAGE line will show an assignment statement with the type of result returned indicated by the variable name. For example:

USAGE: Bytevar = **GETC** [(#Variable)]

shows that the function GETC has one optional argument which must be enclosed in parentheses if given, and returns a function result of type BYTE. The # symbol is used in the USAGE line to emphasize that the optional argument must be the address of the variable, not its value.

Some routines may have more than one optional argument, in which case some or all may be specified. It is permissable to refer to the same LIBRARY routine with different numbers of optional arguments specified in the same program.

PROC ABORT ABORT PROGRAM

USAGE: **ABORT** [Arglist]

ABORT is a procedure which does not return to the calling program but instead exits to the EXECUTIVE. Optionally, it may contain any arguments that are legal for procedure OUTPUT, which will be output to the display.

EXAMPLE 1:

```
INCLUDE LIBRARY
DATA WORD OLDFILENAME = "MYFILE.T"
...
BEGIN
...
ABORT "CAN'T FIND FILE #S", OLDFILENAME
```

will display an error message on the display and abort to the PROMAL EXECUTIVE. See OUTPUT for a description of the arguments which may be used.

FUNC ABS ABSOLUTE VALUE

USAGE: Realvar = **ABS** (Value)

Function **ABS** returns the absolute value of a real number. The function returns type REAL. ABS(X) returns X if X is positive and -X if X is negative.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
REAL X
REAL Y
...
Y = ABS(X-1.)/2.
...
```

NOTE:
1. In PROMAL version 2.0 and earlier, ABS was not included in the the standard LIBRARY, but was in file REALFUNC.S instead. For version 2.1 and later, it is in the LIBRARY for improved convenience, performance, and compatibility with IBM PROMAL.

FUNC ALPHA TEST IF CHARACTER IS ALPHABETIC

USAGE: Bytevar = **ALPHA**(Char)

Function ALPHA returns TRUE if the argument is alphabetic. The argument **Char** is expected to be type BYTE (not a string!). Both upper and lower case letters will return TRUE.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BEGIN
...
CHAR=GETC
IF ALPHA(CHAR)
  ...
```

PROC BLKMOV COPY BLOCK OF MEMORY

USAGE: **BLKMOV** #From, #To, Count

BLKMOV is a procedure for copying a block of memory to another location. **#From** is the starting address. **Count** is the number of bytes to copy. **#To** is the destination starting address. The block being copied can overlap the destination without a problem.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD VALS [200]
BYTE BUF[40]
WORD I
...
BEGIN
...
BLKMOV $0400, BUF, 40
```

copies 40 bytes (decimal) starting at $0400 to BUF.

EXAMPLE 2:

```
BLKMOV #VALS[I], BUF, $8
```

moves 8 bytes (4 words) starting at the Ith word of VALS to BUF. Note the **#** operator before VALS which is required for proper operation.

FUNC CHKSUM COMPUTE CHECKSUM OF BLOCK OF MEMORY

USAGE: Wordvar = **CHKSUM**(#Start, Size)

Function CHKSUM computes the 16 bit checksum of a block of bytes in memory starting at address **Start.** **Size** is the number of bytes to checksum. The returned value is the sum of all the bytes, modulo 65536.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE ARRAY[1000]
WORD ARYCHK
...
ARYCHK = CHKSUM(ARRAY,1000)
...
IF CHKSUM(ARRAY,1000) <> ARYCHK
  PUT NL,"ARRAY has been modified!"
  ...
```

PROC CLOSE CLOSE FILE OR DEVICE

USAGE: **CLOSE** Handle

CLOSE is a procedure which closes a specified word file **Handle.** The argument **Handle** must be the handle for a previously opened file.

EXAMPLE 1:

```
INCLUDE LIBRARY
WORD INPUTFILE      ;File handle
BEGIN
...
INPUTFILE=OPEN(INPUTFILE)
...
CLOSE INPUTFILE     ;Done with file
...
```

It is not normally necessary to close files in a program since all open files will be closed automatically by the EXECUTIVE when exiting from a program. If you are planning to generate stand-alone application programs which will be run without the EXECUTIVE (as described in the PROMAL DEVELOPER'S GUIDE), then you should be careful to close all files, since they will not be automatically closed. Also if you work with several files in a program, it is a good idea to close a file as soon as it is no longer needed, since a limited number of files may be open at once. A power failure or other system failure may leave a file written to disk incomplete unless it has been closed. Be careful not to close a file which you have already closed previously.

FUNC CMPSTR COMPARE STRINGS

USAGE: Bvar = **CMPSTR**(String1, Op, String2 [,Fold [,Limit]])

Function CMPSTR compares two strings. **String1** and **String2** are the addresses of the two strings to be compared, and **Op** is a string (not a character!) specifying which compare operation is desired, chosen from:

"<" "<=" "<>" "=" ">=" ">"

Fold is an optional Boolean (BYTE) argument defaulting to FALSE which, if TRUE, will cause lower case letters to be considered as equal to their upper case equivalents. **Limit** is an optional argument specifying the maximum number of characters to compare in the string, defaulting to 255. The collating sequence for the comparison is the ASCII character set. Two strings are equal if and only if they are the same length and have the same content (or equivalent if Fold is TRUE). If two strings of different lengths match up to the end of the shorter string, the longer string is considered greater.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE LINE[81]
...
GETL LINE
IF CMPSTR (LINE, ">", "M")
```

tests if the string LINE is greater than "M".

EXAMPLE 2:

```
DATA WORD KEYWORD="DRAW","MOVE","ERASE","QUIT",0
...

I=0
REPEAT
  IF CMPSTR (LINE, "=", KEYWORD[I], TRUE, LENSTR(KEYWORD[I])
    ...
  I=I+1
UNTIL KEYWORD[I]=0
```

tests if the string LINE matches the Ith keyword of a table, up to the length of the keyword. (Note: See function LOOKSTR for a better way to do this).

FUNC CURCOL RETURN CURRENT COLUMN OF CURSOR

USAGE: Bytevar = **CURCOL**

Function CURCOL returns the current column number of the text cursor on the screen. The leftmost column is column 0, not column number 1.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
?A
CON MAXCOL = 79  ; Screen width-1, Apple
?
?C
CON MAXCOL = 39  ; Screen width-1, Commodore
?
...
BEGIN
...
IF MAXCOL-CURCOL < LENSTR (STRING) ; won't fit entirely on current line?
  PUT NL
PUT STRING
...
```

FUNC CURLINE RETURN CURRENT LINE NUMBER OF CURSOR

USAGE: Bytevar = **CURLINE**

Function CURLINE returns the current line number of the text cursor on the screen. The topmost line is line 0, not line number 1.

EXAMPLE:

```
INCLUDE LIBRARY
BYTE SAVELOC
...
BEGIN
...
SAVELOC=CURLINE    ;Save line we're on
CURSET 0,0         ;Move to home position
PUT "Error, please try again."
CURSET 0,SAVELOC   ;Back to where we were, col 1.
```

PROC CURSET SET CURSOR POSITION

USAGE: **CURSET** Column, Line

Procedure CURSET sets the screen cursor to a specified column and line. **Column** is the desired column, and **Line** is the desired line number. The home position on the screen (the upper left hand corner of the screen) is location (0,0) not (1,1).

EXAMPLE 1:

```
INCLUDE LIBRARY
BYTE I
...
BEGIN
...
CURSET 0,I
```

moves the cursor to the first column of text row I on the screen.

FUNC DIR EXAMINE DISK DIRECTORY

USAGE: Intvar = **DIR**(Pattern [,Mode])

Function DIR displays the names of any files in a disk directory. For the Apple II, **Pattern** is the desired directory name. No filenames or wildcards are recognized. For the Commodore 64, **Pattern** is a filename string which may include wildcards * and ?. The **Pattern** may optionally have a drive number prefix and a file extension (which can also be a wildcard). The * wildcard matches ANY string and the ? wildcard matches any single character. The function returns an INTeger value indicating the number of files which matched the **Pattern** (including subdirectories for the Apple) if positive or 0, or minus an error code if negative. The absolute value of the error code has the same meaning as for IOERROR for function OPEN. **Mode** is an optional argument, defaulting to 1. If **Mode** is 1 or unspecified, a normal display of file names is made. If **Mode**=0, then the directory will be tested for matching entries and **Intvar** returned, but nothing will be displayed. Alternatively, **Mode** can be an open file handle. In this case, the output is directed to this file or device instead of the screen.

For the Apple II, the format of the output display will be the same as for the FILES command in the EXECUTIVE if the output is to the screen, or will be one filename per line for any other file or device. For the Commodore 64, the format of the output display will be the same as for Commodore BASIC. The pattern matching is performed by the Commodore ROMs resident in the disk drive, and therefore operates as described in the Commodore disk manual. In particular, you should note that a pattern of "*.S" will **not** match all the files ending in ".S", but will instead match ALL the files on the disk. This is because Commodore has chosen to implement the "*" wildcard to mean, "match anything at all" (including ".").

EXAMPLE 1 (for **COMMODORE 64**):

```
INCLUDE LIBRARY
...
BEGIN
...
IF DIR("OLDFILE.D")=1     ; file exists?
  PUT NL,"Want to use existing file?" ...
```

EXAMPLE 2 (for **Apple II**):

```
INCLUDE LIBRARY
...
DATA WORD SUBDIR="ACCOUNTS/" ; Sub-directory in current prefix.
WORD NUMACCTS
...
NUMACCTS=DIR(SUBDIR)  ; Display file name in our sub-directory
...
```

NOTE:
1. For the Apple, any file name part will be ignored. For example, "2:ACCOUNTS/MYFILE.T" is equivalent to "2:ACCOUNTS/".

FUNC DIROPEN **APPLE II ONLY** OPEN DIRECTORY FOR READING

USAGE: Handle = **DIROPEN**(Dirname [, Mode])

Function DIROPEN is used to open a disk directory for reading on the Apple II. **Dirname** is a string specifying the directory name. **Mode** is the optional access mode character, which must be ´R´ (read access) if specified. Opening a directory for writing is not permitted by ProDOS. DIROPEN returns a file handle (type WORD) as in a normal OPEN function, if successful. Once opened, the directory can be read like an ordinary file. Please consult the ProDOS reference manual for information on directory organization.

EXAMPLE 1:

```
...
INCLUDE LIBRARY
INCLUDE PROSYS
DATA WORD PATH = "2:"
WORD DIRHANDLE
...
BEGIN
...
DIRHANDLE = DIROPEN(PATH)
IF DIRHANDLE = 0
  PUT NL,"Can´t open directory for drive 2"
...
```

NOTE:
1. You will need to INCLUDE PROSYS near the beginning of your program in order to use DIROPEN.
2. Although DIROPEN is only available on the Apple, you may open a Commodore 64 directory using the OPEN function (see OPEN).
3. The file PRODOSCALLS.S contains examples of a way to get or set file attributes on the Apple without reading the directory.

FUNC EDLINE EDIT LINE ON SCREEN

USAGE: Bytevar = **EDLINE**(String [,Limit [,Mode [,#Col]]])

Function EDLINE is used to allow on-screen editing of a single line of text in the same way as is supported by the PROMAL EXECUTIVE. **String** is the address of the string to be displayed and edited in place. **String** should be the address of a buffer **large enough to hold at least Limit+1 characters. Limit** is an optional parameter defaulting to 80 which is the maximum number of characters acceptable in the line. **Mode** is an optional argument defaulting to $00 which controls several options based on individual bits in **Mode,** as follows:

```
Bit 0 = 1 (Mode=$01) means display the line in reverse video (highlighted).
        0 means display the line in normal video.
Bit 1 = 1 (Mode=$02) means return "raw" function key codes from the keyboard.
        0 means expand the function keys to their current definitions (see
          FKEYSET).
Bit 2 = 1 (Mode=$04) means return "strange" control keys (explained below).
        0 means ignore "strange" control keys.
Bit 3 = 1 (Mode=$08) means initially display cursor at the column specified
          in the BYTE variable Col, if specified, otherwise at the first
          character.
        0 means initially display cursor after last character.
```

The last optional argument, **#Col,** is ignored unless bit 3 of **Mode** is 1. In this case, **#Col** is the **address** of a variable of type BYTE which contains the desired starting column for the cursor. If the specified column is greater than the length of the line, the cursor will be positioned immediately after the last character. On exit from EDLINE, the variable Col is updated to the position of the cursor at the time of exit from EDLINE.

Mode bits may be combined. For example Mode=$09 enables reverse video and positions the cursor at the start of the field instead of the end (assuming no **#Col** argument is specified).

Function EDLINE returns a byte which is the terminator entered. For normal **Mode,** this will be a carriage return ($0D). However, if bits 1 and/or 2 of **Mode** are 1, it could be a function key, cursor up/down key, control key, etc. "Strange" control keys are defined as those control keys which are not allowed for line editing, or keys returning a value greater than $7F other than function keys. See **Appendix B** for key code values. For the Apple, function keys are defined as either Apple key in conjunction with a number key 1 though 8.

When EDLINE is called, it will display the **String** passed (which can be null), starting at the current cursor position. It will then output enough blanks so that a total of **Limit** characters are displayed. This is particularly useful when bit 0 of **Mode** is set to 1 to select reverse video, since EDLINE will display a reverse video "box" indicating the allowable "field size" on the screen. EDLINE will then position the cursor after the last character of the string (assuming bit 3 of mode is not set) and wait for keyboard input. All line editing keys allowed by the PROMAL EXECUTIVE can be used in the same manner with EDLINE, including CTRL-B to recall a prior line entry. See **Table** 1 of the PROMAL USER'S GUIDE for a complete list of supported editing keys. The **String** will be edited "in place".

Note that during input, the cursor is held "captive" in the limits of the line, making it suitable for various kinds of data entry. By setting **Mode** appropriately, EDLINE can become the basis of an editor or field-oriented data entry system. By setting bits 2 and 3 (Mode=$08+$04) and specifying **#Col**, you can detect, for example, when a "cursor up" key is entered (by the value returned by the function), and what column the cursor was in at the time (by the value returned in Col). You could then call EDLINE again to edit a string which is on the line above the present line on the screen, with the same arguments, and the cursor would appear initially in the same column as on the previous line.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE DUMMY
BYTE BUFFER[81]
...
BEGIN
...
MOVSTR "ERASE ",BUFFER
DUMMY=EDLINE(BUFFER) ; let user complete or change the command
...
```

This program fragment will display the word "ERASE" on the screen, followed by a blank and the cursor. The user could then complete the command as desired.

EXAMPLE 2:

```
INCLUDE LIBRARY

BYTE LINE[41]
BYTE COL
BYTE KEY
...
MOVSTR "This is a line to be edited.", LINE
COL=3
CURSET 8,0
KEY = EDLINE(LINE, 40, $0F, #COL)
...
```

This program fragment will display the specified string starting at the 8th column of the first line on the screen, in a reverse video "box" 40 characters long (which will wrap around to the next line on the Commodore 64), and will position the cursor on the "s" in "This". The line can then be edited by the user in the usual manner. When a "strange" key (such as cursor up or down) is entered, EDLINE returns the edited string in LINE, sets KEY to the key code for the strange key, and updates Col to the cursor position at the time the key was pressed.

NOTE:
1. It is possible, with care, to change almost all of the choices for editing keys for EDLINE, as well as the cursor blink rate. You can also disable CTRL-C (for the Apple) or other editing keys if you wish. See **Appendix G.**

PROC EXIT EXIT FROM PROGRAM

USAGE: **EXIT** [Arglist]

EXIT is a procedure which does not return to the calling program but instead exits to the EXECUTIVE (or to the parent program if this program was LOADed by another). Optionally, the call may contain any arguments that are legal for procedure OUTPUT, which will be output to the display.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BEGIN
...
EXIT "Program Complete."
```

This will display a message on the display and exit to the PROMAL EXECUTIVE. See procedure OUTPUT for a description of what arguments may be used.

PROC FILL FILL A BLOCK OF MEMORY WITH A CONSTANT

USAGE: **FILL** #From, Count [,Byteval]

FILL is a procedure which fills a block of memory with a specified value of type BYTE. **#From** is the desired starting address. **Count** is the number of bytes to fill. The optional argument **Byteval** is the value to be placed in each byte, defaulting to $00. FILL operates much faster than a programmed loop.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BEGIN
...
CON BUFSIZE = 500
BYTE BUF[BUFSIZE]
...
FILL BUF, BUFSIZE
```

zeroes the array BUF of size BUFSIZE bytes.

EXAMPLE 2:

```
BYTE MYSTRING[20]
...
FILL MYSTRING, LENSTR(MYSTRING),' '
```

blank fills the string MYSTRING up to its present length.

When using FILL to zero an array of type WORD or INT, remember that the size needed for the second argument should be twice the array dimension (six times for REAL).

PROC FKEYGET GET A CURRENT FUNCTION KEY DEFINITION STRING

USAGE: **FKEYGET** Keynumber, #String

Procedure FKEYGET sets a string to the currently-defined function key substitution string. **Keynumber** is the desired function key number, from 1 to 8. **#String** is the address of a buffer at least 32 characters long to receive the desired string.

EXAMPLE 1:

```
INCLUDE LIBRARY

BYTE KEYDEF[32]
WORD I
...
BEGIN
...
PUT NL,"The current function key definitions are:"
FOR I=1 TO 8
  FKEYGET I,KEYDEF
  OUTPUT "#C#I = #S",I,KEYDEF
...
```

NOTE:
1. To define function key strings, see FKEYSET.
2. For programs created with the optional GENMASTER utility of the optional Developer's system, function key definitions are initially null strings until defined by calls to FKEYSET.

PROC FKEYSET DEFINE A FUNCTION KEY EXPANSION STRING

USAGE: **FKEYSET** Keynumber, String

Procedure FKEYSET is used to define a function key substitution string of up to 31 characters. **Keynumber** is the desired function key number, 1 to 8. **String** is the desired function key substitution string.

Once the function key substitution string is defined, pressing the function key in the PROMAL EXECUTIVE, or during data entry to a GETL, EDLINE, or INLINE call, will cause the defined string to replace the current line. Up to 31 characters may be defined. Only normal, displayable characters ($20 through $7E) should be used.

EXAMPLE 1:

```
    INCLUDE LIBRARY
    ...
    BEGIN
    ...
    FKEYSET 2,"COMPILE 2:MYPROG"
```

defines function key F2 to be "COMPILE 2:MYPROG".

NOTE:
1. On the Apple II, function keys are activated by holding down either Apple key and pressing a number key 1 through 8.
2. You can ignore function keys in EDLINE/INLINE/GETL by using FKEYSET to set the key definition to a null string (e.g., FKEYSET 1,"").
3. You can cause the function keys to return their original key code in EDLINE/INLINE/GETL by using FKEYSET to define a string consisting of the key code (see **Appendix B**) followed by a zero byte (e.g., FKEYSET 1,"\85" for the Commodore 64 F1 key).
4. Function key settings defined in a program remain in effect when control is returned to the EXECUTIVE.

FUNC GETARGS SPLIT A COMMAND LINE INTO ARGUMENTS

USAGE: Bytevar = **GETARGS**(Argline, #Ptrlist [,Limit [,Sep]])

Function GETARGS is used to parse a line into a list of strings, one string for each argument. **Argline** is the string argument which is to be separated into arguments. **#Ptrlist** is the address of an ARRAY of WORD variables. The function will return a list of pointers to the arguments in this array. **Limit** is an optional argument specifying the maximum number of arguments which may be returned. **Sep** is an optional BYTE argument defaulting to ' ', which specifies what character is to be considered the separator of arguments. The GETARGS function will modify the **Argline** string in place, by installing a 0 byte at the end of each argument, replacing the first separator (usually a blank) after each argument. Each entry in the pointer list will be filled with a pointer to the first **non-blank** character of the argument. The returned value of the function is a BYTE variable indicating the number of arguments returned in the pointer list.

EXAMPLE 1:

```
INCLUDE LIBRARY
BYTE ARGLINE[81]
WORD ARGLIST[8]
...
BEGIN
...
MOVSTR "MYFILE     6 YOURFILE", ARGLINE
...
N = GETARGS(ARGLINE,ARGLIST)
```

will return N=3, and set ARGLIST[0]="MYFILE", ARGLIST[1]="6", and ARGLIST[2]="YOURFILE". In an actual application, ARGLINE would typically be read from the keyboard instead. The effects of GETARGS on the ARGLINE array in memory are illustrated below (**0** indicates $00 terminator):

M	Y	F	I	L	E						6		Y	O	U	R	F	I	L	E	**0**

Before GETARGS call

```
ARGLIST[0]          ARGLIST[1]
                      ARGLIST[2]
```

M	Y	F	I	L	E	**0**					6	**0**	Y	O	U	R	F	I	L	E	**0**

After GETARGS call

NOTE:
1. Since the Argline string is modified in place, if you alter any of the strings returned in ARGLIST, you should be careful not to make them larger or they will affect the content of neighboring strings.

FUNC GETBLKF READ A BLOCK FROM A FILE INTO MEMORY

USAGE: Wordvar = **GETBLKF**(Handle, #Start, Maxsize)

Function GETBLKF does a block read from a file or device. **Handle** is the file handle of the previously opened file (see OPEN for more information on file handles). **#Start** is the desired address where the data should be installed in memory. **Maxsize** is the maximum number of bytes to read. **Wordvar** is returned as the number of bytes actually read. If **Wordvar** is less than **Maxsize**, then end-of-file was encountered before **Maxsize** bytes could be read. GETBLKF does not recognize any record boundaries or delimiters except end-of-file. It is the complementary function to PUTBLKF. It is the fastest way to read data from disk.

EXAMPLE 1:

```
    INCLUDE LIBRARY
    ...
    WORD SCREENFILE     ;File handle
    WORD READSIZE       ;# bytes actually read
    ...
    BEGIN
    ...
    SCREENFILE=OPEN("SCREENDATA.D")  ;Open for reading
    IF SCREENFILE=0     ;Open error?
      ABORT "#CCant read SCREENDATA.D file!"
    READSIZE=GETBLKF (SCREENFILE,$0400,1000)
    ...
```

reads 1000 bytes (decimal) from the file "SCREENDATA.D" into memory starting at location $0400.

NOTE:

1. The predefined (in the LIBRARY) variable DIOERROR can be checked after a GETBLKF operation to test for possible errors, if desired. If DIOERROR=0, the read was completed normally. If DIOERROR = 2, a disk read error occurred (in which case GETBLKF will return as much as could be successfully read before the error).
2. When using GETBLKF to read data into memory starting at a **particular** element of an array, be sure to specify the **#** operator to indicate that you want the address of the array element, not the value. For example,

```
   READSZ = GETBLKF(HANDLE, #BUF[I,0])
```

3. **IMPORTANT: For Commodore 64**, GETBLKF is the only Library routine which uses DYNODISK, if it is enabled. You must **not** mix GETBLKF calls with other, non-DYNODISK read calls (such as GETLF or GETCF) on the same file while DYNODISK is enabled. Also, do not mix GETBLKF calls with DYNODISK off and DYNODOISK on in the same file. To disable DYNODISK from within a program, set C64DYNO to 0 (defined in file PROSYS.S).

FUNC GETC RETURN ONE CHARACTER FROM KEYBOARD

USAGE: Bytevar = **GETC**[(#Variable)]

GETC is a function (not a procedure!) which gets one character from the keyboard and displays it on the screen. It has one optional argument which is the **address** of a variable to receive the character entered. It returns an argument of type BYTE, which is the character read. The same character will be installed in the variable whose address is the argument, if present. The optional argument allows a convenient way to save the character and test it in the same statement. GETC blinks the cursor while waiting for a key to be pressed, and echoes the key to the screen.

CAUTION: If you use the optional second argument, be sure to specify the # operator in front of the variable to receive the character. Otherwise, the character will be installed somewhere in the first page of memory, corresponding to whatever value happens to be in that variable at the time, possibly corrupting the PROMAL system.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE NAME[41]
WORD I
...
BEGIN
...
I=0
WHILE ALPHA(GETC(#NAME[I]))
   I=I+1
NAME[I]=0
```

This fills the buffer NAME with characters from the keyboard until a non-alphabetic character is entered, and terminates it with a $00 byte to make it a string. Alternatively, the form without an argument could be used:

```
I=0
REPEAT
  BUF[I]=GETC
  I=I+1
UNTIL NOT ALPHA(BUF[I-1])
BUF[I]=0
```

NOTE:
1. GETC processes the Alpha lock key (CTRL-A) internally.
2. GETC treats CTRL-Z as end-of-file from the keyboard and therefore returns $00 instead of $1A for CTRL-Z.
3. If you wish to get a key without keyboard echo, see GETKEY.
4. If you wish to test if a key is pressed without waiting for one, see TESTKEY.
5. It is possible to change the cursor blink rate. See **Appendix G.**

FUNC GETCF GET A BYTE FROM A FILE OR DEVICE

USAGE: Flagbyte = **GETCF**(Handle, #Variable)

GETCF is similar to GETC but accepts input from a file or device. The first argument is a WORD variable which is the file **Handle** (see OPEN for information on file handles). The second argument specifies the **address** of the variable to receive the character. GETCF returns FALSE if end-of-file is encountered and TRUE otherwise. **Be sure to remember to specify the # operator on the second argument.**

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE CHAR
WORD INFILE
WORD COUNT
...
BEGIN
...
COUNT=0
INFILE=OPEN(CARG[1])
WHILE GETCF(INFILE,#CHAR)
  COUNT=COUNT + (CHAR=',') ; bump count if is ','
OUTPUT "#C#S contains #W commas.",CARG[1],COUNT
```

This will read the file specified on the command line and display a count of all commas in the file.

NOTE:
1. GETCF is not limited to reading text files. It will correctly return all 256 possible values which can be read from a file, including $00.
2. If the handle is STDIN (the keyboard), then characters are processed as described for GETC above, and GETCF returns TRUE when CTRL-Z is entered.
3. On the Apple after GETCF returns, DIOERR will be 0 normally and 2 if a disk read error occurred, if you wish to check it.

FUNC GETKEY RETURN ONE CHARACTER FROM KEYBOARD WITHOUT ECHO

USAGE: Bytevar = **GETKEY**[(#Variable)]

GETKEY is a function (not a procedure!) which gets one character from the keyboard without displaying it on the screen. It has one optional argument which is the **address** of a variable to receive the value input. It returns an argument of type BYTE, which is the character read. The same character will be installed in the variable whose address is the argument, if present. The optional argument allows a convenient way to save the character and test it in the same statement. **Appendix B** gives the key codes returned by GETKEY.

CAUTION: If you use the optional second argument, be sure to specify the # operator in front of the variable to receive the character. Otherwise, the character will be installed somewhere in the first 256 bytes of memory, corresponding to whatever value happens to be in that variable at the time, possibly corrupting PROMNAL or the operating system.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
PUT NL,"Press any key when ready, or * to exit."
IF GETKEY = '*'
  ABORT "#cProgram terminated."
...
```

NOTE:
1. GETKEY processes Alpha-lock (CTRL-A) internally.
2. GETKEY returns CTRL-Z as $1A, without special treatment.
3. You may alter the cursor blink rate. See **Appendix G.**

PROC GETL GET LINE OF TEXT FROM KEYBOARD

USAGE: **GETL** #Buffer [,Limit]

Procedure GETL inputs a line from the keyboard, allowing all editing (backspace, etc.) supported by the PROMAL EXECUTIVE prior to the carriage return. GETL has one required argument which is the address of the buffer to receive the line. A second optional argument can be used to specify the maximum number of characters to be read. The default limit is 80 characters. The line will be returned as a string, with a $00 byte replacing the carriage return at the end of line. The carriage return is not returned. Therefore the buffer for the default GETL should be 81 bytes long to allow for the full input.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BEGIN
...
BYTE LINE[81]  ;Input line buffer
...
BEGIN
GETL LINE
```

This inputs a line from the keyboard into the LINE buffer.

EXAMPLE 2:

```
BYTE PAGE [41,25]  ; Array of 25 lines of up to 40 chars each
WORD I
...
GETL #PAGE[0,I], 40
...
```

This reads a line form the keyboard into the Ith line of the PAGE array, up to 40 characters long.

NOTE:
1. **Table 1** in the PROMAL LANGUAGE MANUAL lists the line editing keys.
2. Due to buffer size limits, the maximum line size allowable for the Commodore 64 is 80 characters, and 127 characters for the Apple II.
3. It is possible to alter the cursor blink rate and the editing keys used by GETL. See **Appenix G.**
4. GETL always clears a space on the screen large enough to enter Limit characters by outputting blanks from the present cursor position, before accepting input (at the original cursor position). This may cause the screen to scroll if the initial cursor position was within Limit characters of the end of the screen.

FUNC GETLF GET LINE OF TEXT FROM FILE OR DEVICE

USAGE: Flagbyte = **GETLF**(Handle, #Buffer [, Limit])

Function GETLF (not a procedure!) inputs a line from a file or device specified by the file **Handle**, which is the first argument. See OPEN for more information on file handles. The second argument is the address of the buffer to receive the line. An optional third argument can be used to specify the maximum number of characters to be returned. If the line contains more than **Limit** characters, the entire line is read up to and including the carriage return, but only the first Limit characters are copied into the buffer. The line will be terminated by a $00 byte and will not include the carriage return. The returned value of the function is TRUE normally and FALSE (0) if end-of-file was encountered before any bytes could be read from the file or device.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD INPUTFILE
BYTE LINE[41] ; Input buffer
...
BEGIN
...
INPUTFILE=OPEN("MYFILE.T")
IF INPUTFILE=0 ; open error?
  ABORT "#CCan't open MYFILE.T - Program Aborted"
WHILE GETLF(INPUTFILE,LINE,40) ;only 40 chars max please
  PUT NL,LINE
...
```

This will display the first 40 characters of every line of file MYFILE.T.

NOTE:
1. Due to buffer size limitations, a maximum of 127 characters can be read for a line. On the Commodore 64, if the Handle is STDIN (the keyboard), this is reduced to a maximum of 80 characters. To read lines larger than 127 characters from a file, you could use GETCF instead, installing characters in your own buffer one at a time, checking for a carriage return.
2. If the Handle is STDIN (the keyboard), the alpha-lock character (CTRL-A) will be processed internally, and CTRL-Z will be treated as end-of-file if it is the first character of the line.
3. When using GETLF to input starting at a particular element of an array, be sure to specify the # operator to indicate the address of the element. Like all PROMAL routines processing strings, the GETLF procedure expects the **address** of the desired destination for the string.

FUNC GETPOSF **NOT AVAILABLE ON COMMODORE 64** RETURN PRESENT FILE POSITION

USAGE: Wordvar = **GETPOSF**(Handle [,#Segvar])

Function GETPOSF returns the relative position of the next byte to be read/written in a file. **Handle** is the file handle of a previously OPENed file. **Wordvar** is returned as the relative offset from the beginning of the file in bytes, from 0 to 65535. **#Segvar** is an optional address of a word variable to receive the high order 16 bits of the relative offset. It is necessary to specify #Segvar only if the file is more than 64K bytes long and you wish to know the full offset into the file. GETPOSF should not be used for devices.

A common use of GETPOSF is to save the current file position for a file which has been partially read but must be closed temporarily for some reason (such as changing disks during a single-drive copy operation), and then restoring the file to the same position so that you can continue reading.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD CURPOSN
WORD FILE
...
FILE=OPEN("MYFILE.D",R)
...
CURPOSN=GETPOSF(FILE)
CLOSE FILE
...
FILE=OPEN("MYFILE.D",R)
SETPOSF FILE,CURPOSN
...
```

NOTE:
1. This function is not supported on the Commodore 64 because the Commodore hardware and ROMs to not support it.

FUNC GETTST GET T DEVICE STATUS

USAGE: Bytevar = **GETTST** (IOflag)

Function **GETTST** tests if the T device (serial port) is ready to send or receive a character. **IOFlag** is 0 to test the input status and 1 to test the output status. The function returns TRUE if the serial port is ready and FALSE if not. When testing the input status, **GETTST** returns TRUE if a character has been received. When testing the output status, the function returns TRUE if the transmitter is empty (the last character, if any, has been sent).

Appendix F contains additional information on GETTST and related topics.

EXAMPLE 1:

```
INCLUDE PROSYS  ;Where GETTST is defined
...
WORD COM        ; File handle for serial port
BYTE CHAR       ; Received character
...
COM = OPEN("T", 'B') ; Open serial port for input/output
IF COM=0
  ABORT "#CUnable to open serial port"
TDEVRAW=$80    ;Enable "raw" serial input mode (see Appendix F)
...
REPEAT
  IF GETTST(0)    ;Character received from serial port?
    CHAR=GETCF(COM) ;Get it
    PUT CHAR      ;Display it
UNTIL TESTKEY     ;Do it until any key is pressed
CLOSE COM         ;Close the serial port
...
```

NOTE:
1. You will need to have INCLUDE PROSYS near the start of your program in order to use GETTST.

FUNC GETVER OBTAIN PROMAL VERSION CODE

USAGE: Wordvar = **GETVER**

Function GETVER returns a WORD value indicating the version of PROMAL which is running. There are no arguments. The low byte of the returned code is the version number as two hex digits (for example, $21 for version 2.1). The high order byte indicates the target machine for the PROMAL runtime package, as follows: $01 = Commodore 64, $02 = Apple II, $03 = IBM PC small memory model, $04 = IBM PC large code memory model. Additional codes may be defined as PROMAL becomes available on other target machines.

EXAMPLE 1:

```
INCLUDE LIBRARY
INCLUDE PROSYS      ; Where GETVER is defined
...
IF GETVER:> <> $02  ; Make sure we're on an Apple
  ABORT "#cThis program runs only on Apple II"
...
```

NOTE:
1. You will need to INCLUDE PROSYS near the start of your program in order to use GETVER.

FUNC INLINE INPUT LINE OF TEXT FROM SCREEN

USAGE: Bytevar = **INLINE**(String [,Limit [,Mode]])

Function INLINE is the same as EDLINE, except that the **String** to be edited in place is automatically set to null at the start of the routine. The **String** argument should be the address of a buffer **large enough to hold Limit+1 characters.** Please see EDLINE for a full description.

FUNC INLIST SEARCH LINKED LIST

USAGE: Wordvar = **INLIST** (String, Listend [,Fold [,Limit [,Safety]]])

Function INLIST is a special purpose routine for advanced programmers. It searches a linked list of a specific form for an entry matching a string. If the string is not found, 0 is returned. Otherwise, the address of the matching string is returned. **String** is the string desired. **Listend** is a pointer to the end of the list, as shown below (i.e., the link to the first name to try is the word at Listend-2. The optional argument **Fold** is a flag, defaulting to FALSE, which if set to TRUE indicates that lower case alphabetic characters should be considered as matching their uppercase equivalents. **Limit** is an optional argument defaulting to 255, indicating the maximum number of characters required to match in **String**. **Safety** is an optional argument defaulting to 8192 ($2000) indicating the maximum number of entries to test before giving up. **Safety** prevents the function from "hanging up" forever if the linked list is corrupted. The assumed format of the linked list is as follows:

```
                                              <---LISTEND
    -----------------------------------------------
    Address of last entry string (2 bytes)
    -----------------------------------------------
    String for last entry (N bytes)
    -----------------------------------------------
    Address of next-to-last entry string (2 bytes)
    -----------------------------------------------
        ...//...
    -----------------------------------------------
    Address of first entry string (2 bytes)
    -----------------------------------------------
    First Entry string (M bytes)
    -----------------------------------------------
    $0000 (2 bytes, beginning-of-list sentinel)
    -----------------------------------------------
```

EXAMPLE 1:

```
    ;   This example shows how to build a simple Symbol Table for a
    ;   compiler, assembler, etc. using a linked list, where each entry
    ;   is a variable-length name and its associated definition (value).

    INCLUDE PROSYS      ; where INLIST is defined

    WORD SYMTAB [1000] ;space for linked list
    WORD LISTEND       ;ptr to next unused entry

    PROC PUTST ;NAME, VALUE
      ; Install NAME, VALUE into symbol table linked list.
    ARG WORD NAME   ; string to install
    ARG WORD VALUE ; associated definition of name
    WORD STPTR
    BEGIN
    MOVSTR NAME,LISTEND ; install name
    STPTR=LENSTR(NAME)+LISTEND+1 ; after name string
    M[STPTR]=VALUE:<    ; install low byte of value
    M[STPTR+1]=VALUE:>  ; ...hi byte
    M[STPTR+2]=LISTEND:<
    M[STPTR+3]=LISTEND:>
    LISTEND=STPTR+4     ; next available location
    END

    FUNC WORD GETST ; NAME
        ; Returns value stored in symbol table for NAME
    ARG WORD NAME   ; name to look up in symbol table
    WORD ENTRY
    BEGIN
    ENTRY=INLIST(NAME,LISTEND)  ; search list for name
    RETURN (ENTRY+LENSTR(ENTRY)+1)@+  ; = value of NAME
    END
    ...
    ;  Initialize start of list for symbol table...
    SYMTAB[0]=0      ; end of list sentinel
    LISTEND=SYMTAB+2 ; starting address
    ...
```

NOTE:
1. A "real" symbol table manager would need to check for errors such as no more room in the buffer, symbol not found, etc.
2. You will need to have INCLUDE PROSYS near the front of your program to use INLIST.

FUNC INSET TEST IF A CHARACTER IS IN A STRING OR SET

USAGE: Bytevar = **INSET**(Char, String [,Meta])

Function INSET returns the position of a specified character in a string. **Char** is the desired character, **String** is the string to search. **Meta** is an optional argument character, which is usually '-' if specified. If **Meta** is specified, then the **Meta** character can be used to denote a range of characters. **Bytevar** is returned as 0 if the character is not found in the string, or as the index to the matching character **plus one** if the character is found in the string.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE CHAR
BYTE I
...
BEGIN
...
CHAR='A'
...
I = INSET(CHAR, "ABC")
```

This returns I=1 because the A matches first character of the string. When **Meta** is not specified, INSET is often used to find a particular delimiter in a string:

EXAMPLE 2:

```
WORD LINE
...
LINE="100, SPRING INVENTORY"
...
PUT LINE+INSET(',',LINE)
```

This will display:

```
SPRING INVENTORY
```

because the INSET function returns the number of characters to skip over to get beyond the comma. A different use for function INSET is to test for membership of a byte in a set of bytes:

EXAMPLE 3:

```
    BYTE LINE[80]
    ...
    IF INSET(LINE[I], "A-Za-z0-9.",´-´)
```

tests to see if the character LINE[I] is alphabetic, numeric, or a period. The **Meta** argument is specified as ´-´, so "A-Z" will be matched by any character between A and Z inclusive. If LINE[I] was any character between ´B´ and ´Y´ inclusive, INSET would return 2 (the position of the ´-´ plus one).

PROC INTSTR CONVERT SIGNED INTEGER VALUE TO STRING

USAGE: **INTSTR** Value, #Var [,Radix [,Minfield [,Padding]]]

Procedure INTSTR takes a signed value and generates the ASCII string representing the value. **Value** is the desired value to encode and **#Var** is the address of the buffer to receive the ASCII characters. **Radix** is the optional base to be used, defaulting to 10. **Minfield** is the minimum field width to generate, defaulting to 0. **Padding** is an optional character (not string!) argument which is the padding character desired to fill out the buffer to the minimum field width, defaulting to blank.

EXAMPLE 1:

```
    INCLUDE LIBRARY
    BYTE BUF[8]
    INT MYNUM
    ...
    BEGIN
    ...
    MYNUM=568-11
    INTSTR MYNUM, BUF
    ...
    PUT NL,BUF
```

This will display:

```
    557
```

EXAMPLE 2:

```
    INTSTR $FFFE, BUF, 10, 4
```

will install the string " -2" into BUF.

NOTE:
1. If a minimum field width is specified, the number will always be right-justified in the field. If more characters are required to output the number than are specified for the minimum field width, they will be encoded without any error indication.
2. To convert an unsigned (BYTE or WORD) variable, use procedure WORDSTR instead. To convert a REAL value, use procedure REALSTR instead.

PROC JSR CALL MACHINE LANGUAGE SUBROUTINE

USAGE: **JSR** [Address [,Areg [,Xreg [,Yreg [,Flags]]]]]

Procedure JSR calls a machine language subroutine at a specified address, optionally loading the 6502 (or 6510 or 65C02) processor's hardware registers with specified values before the call. **Address** is the address of the desired routine. **Areg**, **Xreg**, **Yreg**, and **Flags** are optional arguments which specify the desired values to be installed in the A, X, Y, and flags (processor status word) registers, respectively. All register arguments should be type BYTE. Naturally the address must be type WORD. It is possible to sample the values returned in the registers from the machine language program.

Please see **Appendix I** for a detailed explanation and examples of JSR.

NOTE:
1. You will need to INCLUDE PROSYS near the beginning of your program in order to use JSR.

FUNC LENSTR RETURN LENGTH OF STRING

USAGE: Bytevar = **LENSTR** (String)

LENSTR is a function which returns a BYTE result indicating the length of the **String** which is the argument.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE NAME[20]
BYTE SIZE
...
BEGIN
...
MOVSTR "Hello", NAME
...
SIZE=LENSTR(NAME)
```

This sets SIZE=5. The size does not include the $00 byte terminator.

NOTE:
1. You may find frequent need of a statement similar to:

```
IF LENSTR(LINE) > 0
  ...
```

where LINE is an array of bytes holding some string. This can be more economically written as:

```
IF LINE@<
  ...
```

which is equivalent, since if a string is non-null, the first character can't be the string terminator.

PROC LOAD LOAD, UNLOAD, OR EXECUTE PROGRAM OR OVERLAY

USAGE: **LOAD** Progname [,Bitflags]

The LOAD procedure loads, unloads, and executes programs and overlays on the Apple II and Commodore 64. **Progname** is the desired program or file name. **Bitflags** is an optional BYTE argument consisting of several 1-bit flags used to control the action taken by the LOADer. Please see the Chapter 8 of the PROMAL LANGUAGE MANUAL for details and examples.

NOTE:
1. You will need to INCLUDE PROSYS near the start of your program in order to use LOAD.

FUNC LOOKSTR SEARCH A LIST OF STRINGS

USAGE: Intvar = **LOOKSTR** (String, Plist [,Nstr [,Fold [, Limit]]])

Function LOOKSTR searches an array of strings, trying to match a given string. **String** is the desired string to try to match, **Plist** is the starting address of a list of pointers to strings, terminated by a $0000 word. **Nstr** is an optional argument specifying the maximum number of strings to search. **Fold** is an optional argument, which if set TRUE, will cause lower case alphabetic characters to be considered equal to their upper case equivalents. **Limit** is an optional argument specifying the maximum number of characters to compare within each string. **Intvar** is returned as -1 if the string did not match, or as the array index to the string that did match.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
INT I
BYTE COMD [20]
DATA WORD KEYWORDS []="MOVE","DRAW","ERASE","DASH","REDRAW","EXIT",0
...
BEGIN
...
MOVSTR "ERASE",COMD
...
I=LOOKSTR(COMD,KEYWORDS)
```

This will return I=2, because the string in COMD matches the third entry in the list.

FUNC MAX RETURN THE LARGEST OF TWO OR MORE VALUES

USAGE: Wordvar = **MAX** (Val1,Val2[,...])

Function MAX returns the largest of two or more arguments **of type WORD (unsigned)**. It is normally used to find the larger of two or more addresses. Do not use it with type REAL arguments.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD I
WORD J
WORD K
...
BEGIN
...
J=1000
K=$D000
...
I=MAX(J,K,$C000)
```

This will return I=$D000.

NOTE:
1. Each value to be tested must be explicitly included in the function call. You cannot find the largest value in an array by merely calling MAX with the array name as an argument. The following example shows how a loop can perform this function.

EXAMPLE 2:

```
WORD LARGEST
WORD MYARRAY[100]
WORD I
...
BEGIN
...
LARGEST = 0     ; Dummy to initialize
FOR I = 0 TO 99 ; Find largest value in array
  LARGEST = MAX (MYARRAY[I], LARGEST)
...
```

FUNC MIN RETURN THE SMALLEST OF TWO OR MORE VALUES

USAGE: Wordvar = **MIN** (Val1,Val2[,...])

Function MIN returns the smallest of two or more arguments **of type WORD (unsigned).** Do not use it with type REAL arguments. It is normally used to find the lesser of two or more addresses.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD I
DATA WORD BOUND []= 100,200,300,400,500,600
...
BEGIN
...
I=351
...
I=MIN(I,BOUND[3])
```

This will return I=351, because 351 is smaller than 400.

FUNC MLGET LOAD MACHINE LANGUAGE PROGRAM

USAGE: Wordvar = **MLGET** (Filename [,Loadaddress])

Function MLGET loads a machine language program. On Commodore 64 systems it is expected to be in standard Commodore format for a machine language PRG file. For Apple II systems it is expected to be a standard Apple BSAVE type file. **Filename** is a string containing the desired file name. **Loadaddress** is an optional load address. If not specified or 0, the load address will be the address at which the program was saved. The function returns a word result which will be $0000 if an error occurred, or the address of the last byte loaded if successful.

You may load as many programs as needed by making multiple calls. No checks are made to see if the loaded program conflicts with other memory usage, and the memory allocation pointers (LOFREE, HIFREE, etc.) are not adjusted. It is your responsibility to insure that an appropriate location is used.

Please see **Appendix I** for more information.

EXAMPLE 1:

```
INCLUDE PROSYS
WORD ENDPROG                              ; Last address
DATA WORD MLPROGNAME = "MLROUTINES"       ; File name
BYTE DUMMY
...
REPEAT
  ENDPROG = MLGET(MLPROGNAME)             ; Load Machine Lang. support routines

  IF ENDPROG = 0
    PUT NL,"Cant load file ",MLPROGNAME
    PUT NL,"Please insert Master diskette and close drive door."
    PUT NL,"Press any key when ready."
    DUMMY = GETC
UNTIL ENDPROG <> 0
...
```

NOTE:
1. For Commodore 64, file names for MLGET must match the desired name **exactly**, including upper and lower case and character set selection.
2. For Apple II, remember that many Apple programs load at $2000 before relocating themselves to their final destination. In this case you may need a BUFFERS HIRES command before loading to help protect PROMAL from being overwritten.
3. You will need to INCLUDE PROSYS near the beginning of your program in order to user MLGET.
4. For Apple II, the default load address is found in the AUX_TYPE field of the directory entry. See the ProDOS Technical Reference Manual for details.

PROC MOVSTR COPY OR JOIN STRINGS OR EXTRACT SUBSTRING

USAGE: **MOVSTR** FromString, ToString [,Limit]

MOVSTR is a procedure which is used to copy strings, to concatenate strings, or to extract substrings (i.e., replaces the LEFT$, MID$, and RIGHT$ functions found in BASIC). **FromString** is the address of the string to copy. **ToString** is the address of the destination. **Limit** is an optional argument specifying the maximum number of characters to copy.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BYTE LINE[81]
BYTE SAVELINE[81]
BYTE KEYWORD[5]
...
BEGIN
...
MOVSTR LINE, SAVELINE
```

This copies the string LINE to the buffer SAVELINE.

EXAMPLE 2:

```
MOVSTR " today.", LINE+LENSTR(LINE)
```

This concatenates the string literal " today." to the end of the string LINE.

EXAMPLE 3:

```
MOVSTR LINE, KEYWORD, 4
```

This extracts the first 4 characters of the string LINE and installs them in the string KEYWORD. The Limit argument does not include the 0 byte string terminator. The destination string may overlap the source string without problems.

EXAMPLE 4:

```
MOVSTR LINE,LINE+1
LINE[0]='A'
```

This inserts the character 'A' at the beginning of the string LINE.

NOTE:
1. MOVSTR always installs a 0 byte terminator at the end of the copied string. Therefore you should always allow room for it.
2. When specifying a particular element of an array for the source or destination , be sure to include the # operator to indicate the address of the element instead of the value (e.g., #BUF[I] is correct).

FUNC NUMERIC TEST IF A CHARACTER IS A DIGIT

USAGE: Bytevar = **NUMERIC** (Char)

Function NUMERIC returns TRUE if the argument is an ASCII numeric digit and FALSE otherwise. The argument is expected to be type BYTE (not a string!).

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD VAL
BYTE CHAR
BEGIN
...
VAL=0
WHILE NUMERIC(GETC(#CHAR))
  VAL=10*VAL+(CHAR-'0')
...
```

This accepts a series of keystrokes until a non-digit is entered, and sets VAL to the numeric decimal value entered.

FUNC ONLINE **AVAILABLE ONLY ON APPLE II** GET VOLUME NAME OF DISK

USAGE: Bytevar = **ONLINE** (Slot, Drive, #Buf)
or
Bytevar = **ONLINE** (0, Unit, #Buf)

Function ONLINE tests if an Apple disk drive (including /RAM disk) is ready, and if so, installs the ProDOS volume name in a specified buffer. In the first form, **Slot** is the Apple slot number (1 to 7), **Drive** is the drive number (1 or 2), and **#Buf** is the address of a buffer of at least 18 bytes which will receive the volume name. The function returns TRUE if the drive is ready and FALSE otherwise (in which case IOERROR holds a code indicating the reason as described in the OPEN function, which will normally be 2 for illegal unit, 3 for not ready, or $28 for non-exisitent). If the first argument is 0, then the second argument is interpreted as a ProDOS unit number (sometimes called a device ID), which is a byte with the following format: Bit 7 is the drive number bit (0=drive 1, 1= drive 2); and bits 4-6 are the slot number (0-7); bits 0-3 are 0. The volume name is returned in the specified buffer as a PROMAL string. The name will have a leading and trailing ´/´, for example "/USER.DISK/".

EXAMPLE 1:

```
INCLUDE LIBRARY
INCLUDE PROSYS
...
BYTE VOLNAM [18]  ; Buffer for diskette volume name
WORD HANDLE
...
IF ONLINE(6,2,BUF) ; have second floppy disk?
  FILE = OPEN ("2:SCRATCH.T",´W´)  ;Open file on drive 2
ELSE
  FILE = OPEN ("1:SCRATCH.T",´W´)
IF FILE=0
  ABORT "#cCan´t open SCRATCH.T for writing"
...
```

NOTE:
1. The /RAM device is normally configured for slot 3 drive 2 and may be tested in with ONLINE.
2. You will need to INCLUDE PROSYS near the front of your program to use ONLINE.

FUNC OPEN OPEN FILE OR DEVICE

USAGE: Wordvariable = **OPEN** (Filename [,Mode [,Nocheck [,Type]]])

OPEN is a function (not a procedure!) which opens a specified file or device for input or output. The first argument is a string which is the desired file or device name. The second argument is optional and is a character (not a string!) specifying the desired access **Mode**, chosen from the following:

´R´	Read access
´W´	Write access
´A´	Append (write, beginning at end of file) access
´B´	Both read and write (**Not available on Commodore 64** except as noted below for use with the command channel or T device).

The default access mode is ´R´. The remaining optional arguments **Nocheck** and **Type** are normally omitted, and are used for opening special system-dependent file types. These system-dependent options are discussed below.

The function OPEN returns a non-zero **file handle** of type WORD if the open was successful, and 0 if it was not. This file handle (also sometimes called a file descriptor) should be saved in a WORD variable. After opening the file, you can refer to the file for I/O operations by simply using this handle. The handle is required as the first argument for other library routines which operate on files.

If OPEN returns 0, indicating that the file could not be opened, then the pre-defined variable IOERROR indicates the reason, as follows:

IOERROR	Meaning **(If function OPEN returns 0)**
0	No error. Normal.
1	Illegal access mode character.
2	Illegal file or device name.
3	Device not ready (or volume not found on Apple II).
4	File not found (for R mode access).
5	File already exists (for W mode access).
6	Can´t open another file (e.g., no more disk buffers) .
7	Write protected (for A or W access).
8 or more	Other (system dependent, see your computer manual).

You should always test for an unsuccessful open.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD INPUTFILE ;input file handle
BYTE LINE[81]  ;input line buffer
BEGIN
INPUTFILE=OPEN("MYDATA.D")
IF INPUTFILE=0
   ABORT "Can't open input file!"
WHILE GETLF (INPUTFILE,LINE)
...
```

The example above could be expanded for better error processing as follows:

EXAMPLE 2:

```
INPUTFILE=OPEN("MYDATA.D",'R')
IF INPUTFILE=0    ; open error?
   CHOOSE IOERROR
   3
     PUT NL,"Disk not ready."
   4
     PUT NL,"MYDATA.D file not found."
   ELSE
     OUTPUT "#CDisk error #I",IOERROR
   ABORT "#CProgram aborted."
...
```

NOTE:
1. The **Commodore 64** firmware limits the maximum number of open files to three, of which only one may be open for writing or appending. If using relative files (see **Appendix M)**, at most one sequential file may be open, and DYNODISK should be off. DYNODISK uses up one buffer inside the 1541 drive.
2. For the **Apple II**, the maximum number of open files is governed by the number of available buffers (see the BUFFERS command in the PROMAL USER'S GUIDE).
3. For the Apple and Commodore, the file handle points to a data structure maintained by PROMAL. The first part of this data structure is the file name, as a PROMAL string. Therefore, if you wish to display the name of a successfully opened file, you can simply output the file handle, for example:

```
PUT NL,"Now reading file ", INPUTFILE
```

4. Devices such as the printer, modem, workspace, etc. are opened in the same manner as files. For example:

EXAMPLE 3:

```
WORD PRTR
...
PRTR = OPEN ("P",'W')
IF PRTR = 0
  PUT NL,"Printer is not ready."
ELSE
  PUTF PRTR, NL,"This will be printed.",NL
...
```

This opens the printer and outputs a line to it. Be sure to always send a final NL to the printer; most printers do not actually print until they receive a carriage return.
5. For special considerations for opening and using the T device (modem), see **Appendix F**. The INTERFACING chapter of the PROMAL LANGUAGE MANUAL contains additional information on opening files and devices, including the printer.

Opening Special System-Dependent Files

The optional argument **Nocheck** is a flag, which, if TRUE, allows files to be opened which do not conform to the standard PROMAL naming conventions and file types. When **Nocheck** is TRUE, you can open any file allowed by the underlying operating system, and no default file extension will be added to the name. This allows your PROMAL programs to read BASIC program files, machine language files, word processor files, etc. If you specify **Nocheck** as TRUE, you may also optionally specify the argument **Type**, which is an argument of type BYTE specifying the type of file desired. This argument is system-dependent.

For the **Commodore 64**, it can be any of the following:

```
'P'  for PRG type files (BASIC and machine language files)
'S'  for SEQ type files (Sequential files)
'U'  for USR type files (User files).
```

The default type is 'S'. For example:

```
C64HANDLE = OPEN("BASIC PROG",'R',TRUE,'P')
```

opens a file named "BASIC PROG" of type PRG for reading.

```
C64HANDLE = OPEN("WordProcData",'W',TRUE,'U')
```

opens a file of type USR for writing.

You can also open to read a directory, open a direct access channel, or the command/error channel. The Type argument should not be specified in this case. For example:

```
C64DIR = OPEN("1:$",'R',TRUE)
```

opens the directory on drive 1. Do not attempt to open a directory for writing. After opening a directory, the contents read will be the sector contents of the directory (minus the track and sector links to the next sector), starting with the BAM. Consult Anatomy of the 1541 Disk, by Abacus Software, for further information on the format of the directory. It is recommended that GETBLKF be used to read the data.

EXAMPLE 4 (COMMODORE 64):

```
WORD C64DA ; Handle for DA file
BYTE CHAN  ; C64 channel # for DA file
...
C64DA = OPEN("#",'B',TRUE)
CHAN = (C64DA+LENSTR(C64DA)+2)@<
```

This opens a direct access file. CHAN is needed so that it can be embedded in the commands to read and write blocks. GETBLKF and PUTBLKF are the best commands to use to read and write the data. Note that the file is opened in 'B' mode, so both reading and writing are permitted. Consult the Commodore 64 Programmer's Reference Guide or the Abacus book for more information.

EXAMPLE 5 (COMMODORE 64):

```
EXT BYTE C64DYNO AT $0DE3 ; From PROSYS.S file
WORD C64CMD  ; File handle for command/error channel
BYTE BUF[81] ; Holds reply from error channel
DATA WORD FMTCMD = "NO:TrashDisk,r8"
...
C64DYNO=0  ;Disable DYNODISK
C64CMD = OPEN("%", 'B',TRUE)
PUTBLKF C64CMD, FMTCMD, LENSTR(FMTCMD)
IF GETLF(C64CMD,BUF)
  PUT NL,BUF,NL
```

This opens the Commodore 64 command/error channel, immediately issues a command to format the disk, and displays the error message from the error channel. Internally, PROMAL will use channel 15 for drive 0: (device 8) and 14 for drive 1: (device 9). Error messages are best read using GETLF. Commands **must** be sent using PUTBLKF (not PUTF or OUTPUTF). Before sending commands to the disk command channel, you should disable DYNODISK, because the commands you send may cause the disk to destroy the special DYNO code in the disk drive. Internally, PROMAL always leaves the command/error channel(s) open all the time; closing and opening an error channel makes the appropriate "connection" through the file handle.

PROMAL assigns Commodore channel 3 to the Printer and 2 to the T device (serial port). The secondary address for the printer can be selected by setting the variable C64PSA before the open (see **Appendix G**). Channel 1 is reserved for the DIR function. Files are assigned channels of 4 and up, with secondary addresses the same as the channel. Therefore if you wish to use a channel for some special purpose in a machine language program, you should choose a channel like 9 or 10 to avoid a possible conflict. Do not close the Commodore command/error channel. Do not attempt any direct serial bus activity from a machine language program with DYNODISK enabled.

Since DYNODISK uses one extra buffer inside the 1541/1571 drive, under some circumstances you may be able to open fewer files with DYNODISK enabled. Under no circumstances should a file be opened with DYNODISK enabled while any other device other than a single disk drive is connected to the serial bus. Failure to observe this precaution will probably result in a "hung" system.

For the **Apple II**, you do not have to specify the **Type** of file in Read mode; any type of file can be opened when **Nocheck** is TRUE. For write mode, the file type should be specified. The values for common ProDOS file types are:

BAD	$01	PCD	$02	PTX	$03	PDA	$04
TXT	$05	BIN	$06	FNT	$07	FOT	$08
BA3	$09	DA3	$0A	WPF	$0B	SOS	$0C
RPD	$10	RPI	$11	DIR	$0F	CMD	$F0
PRML	$F8 (U8)	INT	$FA	IVR	$FB	BAS	$FC
VAR	$FD	REL	$FE	SYS	$FF		

For further information, consult the ProDOS Technical Reference Manual. For example,

```
AIIHANDLE = OPEN("BASPRG",´R´,TRUE)
```

opens the file BASPRG for reading. BASPRG could be a Basic program (or any other type of file).

```
AIIHANDLE = OPEN ("LETTER",´W´,TRUE,$05)
```

This opens file LETTER of type TXT for write access.

For the Apple II, attempting to open a locked file for ´W´ access will be treated as a write-protect error. However, opening a locked file or write protected disk for append mode cannot be detected as an error until an actual attempt is made to write the file. Therefore you should always check DIOERR after the first write operation in append mode. The file PRODOSCALLS.S contains examples which show how to lock or unlock a file.

PROC OUTPUT FORMATTED OUTPUT TO SCREEN

USAGE: OUTPUT Formatstring [, item...]

Procedure OUTPUT displays formatted output on the screen. **Formatstring** is a string which governs how the output will be displayed, and how any optional arguments which follow the format string will be interpreted. The special character **#** is used as a field descriptor inside the format string. Field descriptors indicate what kind of output is desired, chosen from the list below:

#nI	Output signed decimal integer, right justified in a field n characters wide, with leading blank fill. Display "-" after leading blanks if negative (no "+" if plus). If n is omitted, use minimum field width needed to display value.
#nW	Output unsigned decimal word, right justified in a field n characters wide, with leading blank fill. If n omitted, use minimum field width needed to display value.
#nH	Output unsigned hex word, right justified in a field n characters wide, with leading 0 fill. If n is omitted, use minimum field width needed to display value (no leading zeroes).
#nB	Output n blanks (1 if n omitted).
#nS	Output single character or string, left justified in a field of n characters with trailing blank padding.
#nC	Output **one** ASCII character whose value is n decimal. If n is omitted, then output the newline character (ASCII CR, $0D). #0C is not allowed.
#nE	Output scientific notation REAL using a field of n characters (n defaults to 12 if omitted); n must be between 7 and 16.
#n.dR	Output a REAL number using a field n characters wide, with d decimal places shown; n must be between 3 and 12, and d must be less than (n-1).

For each field descriptor in the string there **must** be a corresponding argument following the string (except for #nB and #nC). The value **n** is optional, and defaults to 0 except as noted above. The maximum value for n is 253. Up to a total of 254 characters may be output by the entire procedure call. Hex output will show leading zeros; other numeric output will not. To output the character "#" literally in the format string, use ##.

For #nI, #nW, #nH, and #nS field descriptors, if the value to be output will not fit in the specified field width, extra characters will be output sufficient to display the entire value. For instance, trying to display the value 20000 using a #3W field descriptor will display all five digits, not just 3. However, if fewer digits are needed, blank "padding" will be output to make up the difference. For #n.dR output, remember that you must specify a field width wide enough for the sign and the ".", even if you **know** the answer will be positive (a blank will be displayed). If you try to output a value using #n.dR which is too large to be displayed, PROMAL will first try to display the number using #nE format instead (with the same n as you specified). Failing that, it will print asterisks instead of a value. It is usually a good idea to pick a larger value for n than you really think you will need when using #n.dR format output.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD N
...
BEGIN
N=723
...
OUTPUT "The answer is #W days.", N
```

This will output to the display:

```
The answer is 723 days.
```

EXAMPLE 2:

```
BYTE LINE[81]
LINENO=20
MOVSTR "BEGIN",LINE
...
OUTPUT "#C#4H#5B#S",LINENO,LINE
```

will display (after a carriage return):

```
0014     BEGIN
```

EXAMPLE 3:

```
INCLUDE LIBRARY
...
REAL X
DATA REAL PI = 3.1415926535
...
X = PI * 100000. / 3.
OUTPUT "PI=#10.4R, X=#13E", PI,X
```

will display:

```
PI=    3.1416, X= 1.047198E+06
```

NOTE:

1. The **format string is always required**, and **the number of arguments after the format string must agree with the number of field descriptors given in the format string** (excluding #nB and #nC). You may **not** simply OUTPUT variable names without a format string to display their value!

2. You may output single characters (type BYTE) as well as strings (type WORD) using the #nS field descriptor.

3. The output forms for REAL output will display with rounding based on digits beyond the displayed field. However some decimal fractions such as .005 are not exactly representable in binary format (so, for example, .005 is really .00499999999...). Therefore a value of exactly .005 may be displayed as .00 instead of .01 with a #n.2R field specification.

4. Some useful forms of the #nC field descriptor are:

#C or #13C	Start a new line (carriage return)
#12C	Clear the screen and home the cursor
#15C (Apple) or #18C (Commodore)	Start reverse video
#14C (Apple) or #146C (Commodore)	End reverse video

More information on formatted output is given in the INTERFACING chapter of the PROMAL LANGUAGE MANUAL. The BUDGET.S demo program on the PROMAL disk illustrates how to use formatted output for preparing tabular output data.

PROC OUTPUTF — FORMATTED OUTPUT TO FILE OR DEVICE

USAGE: **OUTPUTF** Handle, Formatstring [, item...]

Procedure OUTPUTF operates in the same manner as procedure OUTPUT above, except that the first argument must be a file **Handle** of a previously opened file or device, which is to receive the output.

EXAMPLE 1:

```
    INCLUDE LIBRARY
    ...
    WORD I
    BEGIN
    ...
    I=100
    ...
  PRTR = OPEN ("P",'W')
  ...
  OUTPUTF PRTR,"#C#10B#I days.#C", I
```

This will output to the printer:

```
          100 days.
```

A carriage return will be written after the line, because of the #C field specified at the end of the **Formatstring**. Note that it is important to remember to send a final CR to the printer, because most printers accumulate characters in a buffer until a carriage return is received. If no final CR is received, the last line will never be printed.

EXAMPLE 2:

```
  WORD OUTFILE    ; Output file handle
  REAL NETWORTH   ; Total net worth in $
  ...
  OUTFILE=OPEN(CARG[1], 'W') ; Open specified output file
  IF OUTFILE=0
    ABORT "#cUnable to open output file #S",CARG[1]
  ...
  OUTPUTF OUTFILE, "#cYour current net worth = $#8.2R", NETWORTH
  ...
```

NOTE:
1. More information on output to files and devices is given in the INTERFACING chapter of the PROMAL LANGUAGE MANUAL. More examples of output formatting are given for PROC OUTPUT, above.
2. On the Apple II, you may test DIOERR if you wish after writing to a file. DIOERR is set to 0 normally, 1 if the disk is full, and 3 for a disk write error.

PROC PROQUIT EXIT FROM PROMAL SYSTEM

USAGE: **PROQUIT**

Procedure PROQUIT causes an immediate exit from the PROMAL environment. For the Commodore-64, the computer is reset, re-starting BASIC. For the Apple II, the ProDOS "Quit" call is executed as described in Apple's ProDOS Technical Note #7, which will result in a prompt for a new path name and complete prefix for the next system program to be executed. PROMAL does not close any files prior to exiting. However, for the Apple II, PROMAL will restore the /RAM disk using the Apple-presecribed method if it was disabled on startup.

EXAMPLE 1:

```
INCLUDE PROSYS
...
PROQUIT  ; Permanently exit PROMAL
```

NOTE:
1. You will need to INCLUDE PROSYS near the beginning of your program in order to use PROQUIT.
2. Once you exit PROMAL, it must be re-booted to resume. There is no "warm" entry point.

PROC PUT OUTPUT CHARACTERS OR STRINGS TO THE SCREEN

USAGE: **PUT** item [, item...]

PUT is a procedure for outputting text (including control characters) to the display. PUT may have one or more arguments. Each argument may either be a single character or the address of a string.

EXAMPLE:

```
PUT "Hello, world!",13
```

outputs the string "Hello world!" followed by a carriage return (13 decimal). No carriage return is automatically added before or after the PUT is executed; it must be explicitly indicated. This allows lines of output to be generated using as many separate PUTs as needed.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
WORD PHRASE1
...
PHRASE1= "Abe Lincoln"
PUT cr," The answer was ",PHRASE1, " or "
PUT "Harold Robbins."
```

This will output the sentence:

The answer was Abe Lincoln or Harold Robbins.

on a new, single line.

NOTE:
1. PUT treats any argument between $00 and $FF inclusive as a single character to be output, and all other values as a pointer to a string of characters to be output. Strings must be terminated by a $00 byte.
2. You may **not** use PUT to display the value of numeric values. Use OUTPUT to perform this function.
3. If you wish to output a **string** starting at a particular element of an array of bytes, don't forget the **#** operator (for example, PUT #PAGE[0,I]). Otherwise, only a single character will be printed (for the reason given in note 1 above).
4. PUT 12 will clear the screen. PUT $12, X, $92 will output X in reverse video on the Commodore 64. PUT $0F, X, $0E will output X in reverse video on the Apple II.
5. PUT N can be used to change text colors on the Commodore 64, where N is as follows: $05=WHT, $1C=RED, $1E=GRN, $1F=BLU, $81=ORG, $90=BLK, $95=BRN, $96=LTRED, $97=GRY1, $98=GRY2, $99=LTGRN, $9A=LTBLU, $9B=GRY3, $9C=PUR, $9E=YEL, $9F=CYN.
6. More information about PUT is contained in the INTERFACING chapter of the PROMAL LANGUAGE MANUAL.

PUTBLKF WRITE MEMORY BLOCK TO FILE OR DEVICE

USAGE: **PUTBLKF** Handle, #Start, Size

Procedure PUTBLKF does a block write to a file or device. **Handle** is the file handle of the previously opened file. **#Start** is the address of the first byte to be written. **Size** is the size of the block to be written, in bytes. The output will be an exact match of the contents of the memory block; no conversions take place, and no terminators or delimiters are added. PUTBLKF and GETBLKF may be used to save and restore memory images, such as arrays or buffers or complete screens.

EXAMPLE:

```
INCLUDE LIBRARY
...
WORD OUTFILE
BYTE BUFFER[300]
...
BEGIN
...
OUTFILE=OPEN(CARG[1],'W')  ;open file name given on command line
IF OUTFILE=0
   ABORT "Can't open file!"
PUTBLKF OUTFILE,BUFFER,300 ;save buffer contents to file
...
```

writes the contents of the BUFFER array to the file specified as the first argument on the command line.

EXAMPLE 2:

```
INCLUDE LIBRARY
...
CON REALSZ = 6           ; # bytes for each REAL variable
REAL ELASTICITY [100]    ; Elasticity matrix for stress analysis
WORD TEMPFILE            ; Temporary file
WORD I                   ; Index to ELASTICITY matrix
...
TEMPFILE=OPEN ("TEMPFILE.MEM", 'w')
...
PUTBLKF TEMPFILE, #ELASTICITY[I], REALSZ*(100-I) ; Save end of matrix
...
```

This saves an exact memory image of the REAL values ELASTICITY[I] through ELASTICITY[99] inclusive to file TEMPFILE.MEM. No conversion to ASCII takes place (i.e., TEMPFILE.MEM is not a text file).

NOTE:
1. If you wish, you may test DIOERR after a PUTBLKF to check for disk errors. DIOERR=0 normally; 1=disk full (works for W device too); 3=disk write error.

PROC PUTF OUTPUT CHARACTERS OR STRINGS TO FILE OR DEVICE

USAGE: **PUTF** Handle, item [,item...]

Procedure PUTF is similar to PUT except the first argument must be a file **Handle** for a previously opened file or device.

EXAMPLE:

```
INCLUDE LIBRARY
...
WORD OUTFILE
DATA WORD FILENAME = "1:MYFILE.D"
BYTE LINEOBUF[20]
BYTE LINE[81]
...
BEGIN
...
OUTFILE=OPEN(FILENAME,'W') ;File name specified in DATA stmt.
...
PUTF OUTFILE,LINENOBUF,' ',LINE,NL
```

This outputs the string LINENOBUF, a blank, the string LINE, and a carriage return to the output file MYFILE.D on drive 1.

NOTES:
1. See PUT above for more information about valid arguments.
2. More information on using PUTF to output to files or devices (including the printer) is given in Chapter 6 of the PROMAL LANGUAGE MANUAL.

FUNC RANDOM RETURN A RANDOM VALUE OF TYPE WORD

USAGE: Wordvar = **RANDOM** [(Seed)]

Function RANDOM returns a pseudo random number **of type WORD**, uniformly distributed between 1 and 65535. If the optional non-zero argument Seed, of type WORD is specified, it will be used as the seed to generate this and any succeeding random numbers.

EXAMPLE 1:

```
INCLUDE LIBRARY
WORD DIEROLL
...
BEGIN
...
DIEROLL = RANDOM % 6 + 1
```

This sets DIEROLL to a random number between 1 and 6 inclusive.

NOTE:

1. RANDOM uses a fast, feedback-shift-register method for generating random numbers, suitable for games, etc. It does not generate random numbers of type REAL.

PROC REALSTR CONVERT REAL VALUE TO STRING

USAGE: **REALSTR** Realval, #Buffer, Fieldwidth [,Decplaces]

Procedure REALSTR is used to convert a REAL numeric value to an ASCII string representing its value. **Realval** is the desired value to convert. **#Buffer** is the address of the string to receive the ASCII numeric representation. **Fieldwidth** is the desired number of characters to represent the number. **Decplaces** is an optional argument specifying the desired number of decimal places to be displayed. If **Decplaces** is omitted, the number will be converted using scientific notation. **Fieldwidth** and **Decplaces** should be expressions of type BYTE or WORD.

Fieldwidth must be specified between 3 and 12 if **Decplaces** is specified (for normal output), or between 7 and 16 if **Decplaces** is not specified (for scientific notation output). If **Decplaces** is specified, it must be less than or equal to the field width minus two. This is because the field width must always include room for a sign and the decimal point itself. If the sign of the value to be printed is +, a blank will be output instead. If the sign of the value is negative, a ´-´ will be output immediately to the left of the leftmost digit (with any necessary blank padding).

If **Decplaces** is specified, but the value is too large to fit in the specified format, REALSTR will first attempt to convert the number in scientific notation in the specified field width. If it is still too large, the number will not be printed, and the field will be filled with asterisks (*).

EXAMPLE 1:

```
INCLUDE LIBRARY
REAL COST
REAL OVERHEAD
REAL PROFIT
REAL GROSS
BYTE BUFFER[10]
BEGIN
GROSS=1299.95
COST=557.44
OVERHEAD = .18*GROSS
PROFIT=GROSS-COST-OVERHEAD
...
REALSTR PROFIT,BUFFER,7,2
...
PUT NL,"Our profit = $",BUFFER
```

This will display:

```
Our profit = $ 508.52
```

EXAMPLE 2:

```
REAL XVAL
...
XVAL=-0.0000005543
...
REALSTR XVAL,BUFFER,12
```

This will install "-5.54300E-07" in BUFFER.

If the format of the output from REALSTR does not exactly meet your needs, it is usually simple to write a procedure to manipulate the converted output into the format you do want. For example, the following program fragment will pad BUFFER with leading asterisks, such as might be used in a program to write checks:

```
WORD PRINTER ;File handle
REAL AMOUNT
BYTE BUF[10]
WORD I
...
PRINTER=OPEN("P",W)
...
AMOUNT=887.50
...
REALSTR AMOUNT,BUF,9,2
I=0
WHILE BUF[I]=' '
  BUF[I]='*'
  I=I+1
PUTF PRINTER,'$',BUF
...
```

This would print:

```
$***887.50
```

PROC REDIRECT REDIRECT INPUT OR OUTPUT

USAGE: **REDIRECT** #STDIN [,Handle]
- or -
REDIRECT #STDOUT [,Handle]

Procedure REDIRECT is used by advanced programmers to redirect one of the two standard I/O paths available in PROMAL: STDIN (standard input), or STDOUT (standard output). Each of these paths is a global variable of type WORD, defined in LIBRARY, and is initialized to point by default to the keyboard device for input or the screen device for output. **Handle** is a file handle of a previously opened file or device. The REDIRECT procedure sets the standard path to point to the open file or device. If the **Handle** argument is not given, the default redirection is made back to the keyboard or screen. If the handle is specified, it must be open and must have the appropriate mode (direction) for the specified STDxxx (e.g., you can't redirect STDOUT to the keyboard). A

violation of either of these requirements generates a runtime error. Only the two global variables above can be redirected. The EXECUTIVE will automatically redirect STDIN and STDOUT back to the default devices at program termination.

EXAMPLE 1:

```
    INCLUDE PROSYS   ; Where REDIRECT is defined
    ...
    WORD OUTFILE
    ...
    OUTFILE=OPEN("SCREENFILE.T",'W')
    REDIRECT #STDOUT,OUTFILE
    ...
    PUTF STDOUT,"This will go to SCREENFILE.T",cr
```

NOTE:
1. You will need to INCLUDE PROSYS near the beginning of your program to use REDIRECT.
2. This function is used by the EXECUTIVE to redirect input and output.

FUNC RENAME RENAME A FILE

USAGE: Bytevar = **RENAME** (Oldfile, Newfile)

Function RENAME is used to change the name of an existing file. **Oldfile** is a string specifying the old file name, as described for OPEN. For the Commodore 64, it may optionally include a drive prefix and file extension. For the Apple II, it may optionally include a drive prefix and pathname. **Newfile** is a second string specifying the desired new name, which must be unique. If the drive or prefix is specified for **Newfile,** it is ignored, and the drive number or prefix for **Oldfile** will be used. It can change the file extension, however. The function returns 0 normally or an error code as described for OPEN.

EXAMPLE 1:

```
    INCLUDE LIBRARY
    ...
    BYTE RENAMERROR
    ...
    BEGIN
    ...
    RENAMERROR=RENAME("TEMP",CARG[1])
    IF RENAMERROR
       PUT NL,"Attempt to rename TEMP.C to ",CARG[1]," failed."
    ...
```

NOTE:
1. For the Commodore 64, a default file extension will be applied unless NOFNCHK (Defined in file PROSYS.S) bit 7 is 1 (set to $80). If NOFNCHK bit 7 is set, non-SEQ files or files with lower case letters can be renamed. Normally, NOFNCHK is 0, which matches only upper case file names and applies a default extension if none is specified.

2. For the Apple, setting bit 7 of NOFNCHK will allow renaming a file or subdirectory with no file extension.

PROC SETPOSF **NOT AVAILABLE ON COMMODORE 64** SET FILE POSITION

USAGE: **SETPOSF** Handle, Position [,Segment]

Procedure SETPOSF sets the relative position of the next byte to be read/-written in a file. **Handle** is the file handle for a previously opened file. **Position** is a WORD value giving the desired file position. If the desired file position is greater than 65535 (64K), then **Segment** should be specified as the high order 8 bits of the complete 24 bit file position. The first byte of the file is byte 0. If the position specified is greater than the current end-of-file, then the file will be positioned to end-of-file instead, without any error indication. Therefore if you wish to use SETPOSF for implementing a random-access file organization, you should initialize the file when it is created by writing dummy records to the file until it has reached the desired maximum size.

A common use of SETPOSF is to determine a file's size. To do this, open the desired file, then use SETPOSF to set the file to a position known to be larger than end-of-file. Then use GETPOSF to read the true end of file position. An example for function GETPOSF, above, illustrates a second common use of SETPOSF.

EXAMPLE 1:

```
INCLUDE LIBRARY
CON RECSIZE=80
BYTE RECORD[RECSIZE+1] ; Current record contents
...
PROC GETRECORD ; File, RecNum
  ; Read record # RecNum from File into Record
ARG WORD FILE ; Open file handle
ARG WORD RECNUM ; Desired record #
BEGIN
SETPOSF FILE,RECNUM*RECSIZE
IF GETBLKF(FILE,#RECORD,RECSIZE) < RECSIZE
  PUT NL,"***Tried to read beyond end of file on file #S",FILE
  OUTPUT "#C***Record requested = #W",RECNUM
  CLOSE FILE
  ABORT "#CFile closed, program terminated."
END
...
```

The above example shows a routine to read a random record into memory from a database file, given the record number and file handle, for a file of up to 64K bytes.

FUNC SETPREFIX **NOT AVAILABLE ON COMMODORE 64** SET PATHNAME

USAGE: Bytevar = **SETPREFIX**(Dirname)

Function SETPREFIX attempts to set the current pathname to the specified volume or directory name string, **Dirname**. If successful, it returns TRUE. If the specified directory is not on line, FALSE is returned and the current path remains unchanged. The string specified by **Dirname** must end with a /. If a leading / is not specified, the Dirname will be appende to the current prefix.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
DATA WORD VOLNAME = "/ACCOUNTS/"  ;Desired volume name
...
IF SETPREFIX(VOLNAME)
  RECEIVEABLES
  ...
ELSE
  PUT NL,"Can´t find ", VOLUME," disk"
...
```

FUNC STRREAL CONVERT NUMERIC STRING TO REAL VALUE

USAGE: Bytevariable = **STRREAL** (String, #Variable)

STRREAL is a function which decodes (converts) a string into a numeric value of type REAL. The first argument is the address of the desired string. The second argument is the **address** of the REAL variable to receive the value represented by the string. The string may have any number of leading blanks and optionally a leading minus sign (-). The string may express the number in normal notation or scientific notation (E-format). Conversion proceeds until a character is encountered which cannot legally be part of the number (such as a trailing blank, end-of-line, comma, etc). The function returns a result of type BYTE which is an index to this delimiter. A returned value of 0 indicates no legal digits were encountered, probably indicating an error condition.

EXAMPLE:

```
INCLUDE LIBRARY
BYTE LINE[81]
REAL VELOCITY
BYTE INDEX
...
BEGIN
...
GETL LINE
INDEX=STRREAL(LINE, #VELOCITY)
IF VELOCITY < 0.0
  ...
```

This would read a line from the keyboard and install the value of the number typed into the variable VELOCITY. Some examples of acceptable input are shown below:

```
0     3.14     9070     .077     7856.004     -200000    -5.56333308E-11
```

CAUTION: Be sure to remember to specify the # operator in front of the REAL variable to receive the value.

A general purpose numeric input routine, INPUTR, is described in **Chapter 5** of the PROMAL LANGUAGE MANUAL and is provided on disk file INPUTR.S.

FUNC STRVAL CONVERT NUMERIC STRING TO WORD OR INT VALUE

USAGE: Bytevar = **STRVAL** (String, #Variable [,Radix [,Maxfield]])

STRVAL is a function which decodes (converts) a string into a numeric value. STRVAL may have two to four arguments. The required arguments are **String**, the desired string, and **#Variable**, the **address** of a WORD or INT variable (**not BYTE or REAL!**) to receive the value represented by the string. **Radix** is an optional conversion base defaulting to base 10, and **Maxfield** is an optional maximum field width defaulting to 255 characters. The value to be converted may be signed or unsigned. The string may have any number of leading blanks. Conversion proceeds until **Maxfield** characters are used from the string or until a character is encountered which cannot legally be a digit in a number in the specified or default base. A byte variable is returned as an index to this delimiter.

EXAMPLE 1:

Assume that the following program segment inputs the line, " 123,456" from the keyboard (without the quotes):

```
INCLUDE LIBRARY
...
BYTE LINE[81]
WORD XDIST
WORD YDIST
...
BEGIN
GETL LINE
BINDEX = STRVAL (LINE,#XDIST)
```

This will install the value 123 decimal in XDIST and set BINDEX=4. If desired, additional statements could determine that the delimiter was "," and so decode any additional values (such as the 456):

EXAMPLE 2 (continued from Example 1 above):

```
IF LINE[BINDEX]=',' 
  BINDEX=STRVAL (LINE+BINDEX+1, #YDIST)
```

EXAMPLE 3:

Assume BUF contained "BDF30A". Then:

```
BINDEX=STRVAL(BUF,#PC,16,3)
```

would set PC to $0BDF and return BINDEX=3, because a maximum field width of three characters was specified. Base 16 decoding was specified. Any radix between 2 and 36 can be used.

NOTES:
1. Be sure to remember to specify the # operator in front of the variable name to receive the numeric value. If you forget it, the value will be installed in memory at whatever address happens to be in that variable at the time!
2. If you wish to input a number of type BYTE, first use STRVAL with a destination of type WORD, and then copy the low byte to the final destination. If you use STRVAL to decode directly to a BYTE variable, the following byte in memory will also be affected.
3. If you wish to input a number of type REAL, use function STRREAL.
4. If the function returns 0 (no digits), the variable is also set to 0.
5. If frequent numeric input is anticipated from the keyboard, you may wish to use the following function (which can be found as file INPUTW.S on a PROMAL disk), which displays a specified prompt and returns a WORD value typed from the keyboard:

```
FUNC WORD INPUTW ; Prompt
  ; Output PROMPT, accept line of numeric input from keyboard,
  ; return the numeric value.
ARG WORD PROMPT   ; Desired prompting message
WORD TEMP         ; Temporary value
BYTE INDEX        ; Number of digits input
OWN BYTE BUF[10]  ; Buffer for keyboard input
BEGIN
REPEAT
  PUT NL,PROMPT   ; Display prompt
  GETL BUF,10     ; Input line
  INDEX=STRVAL(BUF,#TEMP)  ; Convert to numeric value
  IF INDEX=0      ; Invalid entry?
    PUT NL,"Please enter a numeric value."
UNTIL INDEX > 0
RETURN TEMP
END
```

The following example illustrates the use of this function:

```
WORD ILINE
WORD MAXLINE
...
BEGIN
ILINE = INPUTW("What line do you wish to go to? ")
IF ILINE > MAXLINE
  ...
```

FUNC SUBSTR LOCATE SUBSTRING IN STRING

USAGE: Bytevar = **SUBSTR**(Wantstring, Trystring [,Fold [,Max [,Limit]]])

Function SUBSTR searches a string **Trystring** for the presence of another string, **Wantstring**. **Fold** is an optional argument defaulting to FALSE, which, if TRUE, causes lower case letters to be treated as matching upper case letters. **Max** is an optional argument specifying the last character position in **Trystring** at which the match can start, defaulting to LENSTR(Trystring). For instance, if **Max** is 1, then the match must occur starting with the first character of **Trystring**. **Limit** is an optional argument specifying the number of characters in **Wantstring** which must match, defaulting to LENSTR(Wantstring). For example, if **Limit**=2, then SUBSTR will consider a match made if the first two letters of **Wantstring** are found in **Trystring**. The function returns zero if no match is found, or an index to the character **plus one** if it is found.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
DATA WORD ASTRING = "PROMISE ME PROMAL FOR MY BIRTHDAY"
DATA WORD WSTRING = "PROMAL"
BYTE TRY1
BYTE TRY2
BYTE TRY3
...
TRY1=SUBSTR(WSTRING,ASTRING)
TRY2=SUBSTR(WSTRING,ASTRING,TRUE,20,4)
TRY3=SUBSTR(WSTRING,ASTRING,TRUE,10)
```

will set TRY1 to 12, TRY2 to 1, and TRY3 to 0.

FUNC TESTKEY TEST IF A KEY IS PRESSED

USAGE: Bytevar = **TESTKEY** [(#Char)]

Function TESTKEY tests if a key is pressed on the keyboard. If not, it returns 0. If a key is pressed, it is returned as the value of the function and also is installed in the optional character address if specified. The character is not echoed to the display. The key code returned will be ASCII as given in **Appenidx B.** Testkey does not display or blink the cursor.

EXAMPLE 1:

```
INCLUDE LIBRARY
...
BEGIN
...
REPEAT
  NOTHING
UNTIL TESTKEY
```

This waits for any key depression without echoing it to the screen.

NOTE:
1. **CAUTION:** be sure to remember to specify the # operator in front of the variable name to receive the key value.
2. The Commodore 64 "Kernal" ROM software does not support ongoing keydown detection. Therefore calling TESTKEY in a loop will not return another non-0 result until the previous key is released on the Commodore. However, the space bar will auto-repeat at about 10 "hits" per second.
3. For the Apple II, all keys auto-repeat after a brief pause.

FUNC TOUPPER CONVERT LOWER CASE CHARACTER TO UPPER CASE

USAGE: Bytevar = **TOUPPER** (Char)

TOUPPER is a function which takes a single argument of type BYTE and returns an argument of type BYTE. If the argument is a lower case letter, the returned value is the upper case equivalent; otherwise, the argument is returned unchanged.

EXAMPLE 1:

```
    INCLUDE LIBRARY
    ...
    PUT NL,"Do you wish to accept your mission? "
    IF TOUPPER(GETC)='Y'  ; accept 'y' or 'Y'
      TAKEMISSION
    ...
```

EXAMPLE 2:

```
   FUNC WORD UPPERSTRING ; String
     ; Convert all lowercase chars to uppercase in string.
     ; Return same string, updated in place.
   ARG WORD STRING      ; String to convert to uppercase
   WORD I               ; Address of character in string
   BEGIN
   I=STRING             ; Addr of 1st char of string
   WHILE I@<            ; Not end of string
     M[I]=TOUPPER(I@<)  ; Convert if is lowercase
     I=I+1              ; Address of next char
   RETURN STRING
   END
```

NOTE:
1. The argument for TOUPPER must be a single character, not a string.

PROC WORDSTR CONVERT UNSIGNED VALUE TO STRING

USAGE: **WORDSTR** Value, #Buf [,Radix [,Minfield [,Padding]]]

Procedure WORDSTR is the inverse function of STRVAL. It takes an unsigned value and generates the ASCII string representing the value. **Value** is the desired value to encode and **#Buf** is the address of the buffer to receive the ASCII characters. **Radix** is the optional base to be used, defaulting to 10. **Minfield** is the minimum field width to generate, defaulting to 0. If a minimum field width is specified, the number will always be right-justified in the field. If more characters are required to output the number than are specified for the minimum field width, they will be encoded without any error indication. **Padding** is an optional character (not string) argument which is the padding character desired to fill out the buffer to the minimum field width, defaulting to blank.

EXAMPLE 1:

```
INCLUDE LIBRARY
BYTE BUF[8]
BEGIN
ADDR=$FFFF-1
...
WORDSTR ADDR, BUF
```

This will install the string "65534" into BUF.

EXAMPLE 2:

```
WORDSTR $BD, BUF, 16, 4, '0'
```

This will install "00BD" into BUF.

NOTE:
1. If you wish to convert a real number, use procedure REALSTR. To convert a signed integer, use procedure INTSTR.

FUNC ZAPFILE DELETE FILE

USAGE: Bytevar = **ZAPFILE** (Filename [,Wildflag])

Function ZAPFILE deletes a file (or optionally, a group of files). **Filename** is a string argument specifying the file to be deleted. For the Commodore 64, it can have an optional drive number prefix. For the Apple II, it may have a pathname. The optional argument **Wildflag** is a byte value defaulting to FALSE. If TRUE, the **Filename** argument can include the wildcard characters ? and *. In this case, all files matching the pattern will be deleted. Wildcards are not supported for the Apple II. The function returns 0 if successful and an error number as described for OPEN if not. However, an attempt to delete a file which does not exist is not considered to be an error, because this is the way the Commodore ROMs work. If you wish to flag an

attempt to delete a non-existent file as an error, you can do it by first doing a DIR to determine if it exists and issuing an error message if it doesn´t. For the Apple II, ZAPFILE can be used to delete a subdirectory, provided it has no files left in it (see note 2 below).

EXAMPLE:

```
INCLUDE LIBRARY
BYTE ZAPERROR
...
BEGIN
...
ZAPERROR=ZAPFILE(CARG[1])
CHOOSE ZAPERROR
0
  PUT NL,CARG[1]," deleted."
2
  PUT NL,CARG[1]," is not a legal file name."
7
  PUT NL,"Not deleted, disk is write-protected."
ELSE
  PUT NL,"Not deleted, error."
...
```

NOTES:
1. For Commodore 64, if you want to delete a file which is not type SEQ, does not have a file extension, or has any lower case letters, you will have to set the NOFNCHK flag to $80 (defined in file PROSYS.S).
2. For the Apple II, setting NOFNCHK=$80 will allow file names with no extension or empty subdirectories to be deleted. NOFNCHK is defined in file PROSYS.S

APPENDIX A

ASCII CHARACTER SET TABLE

Hex	Dec.	Char.	Hex	Dec.	Char.	Hex	Dec.	Char.	Hex	Dec.	Char.
00	0	NUL	20	32	space	40	64	@	60	96	`
01	1	SOH	21	33	!	41	65	A	61	97	a
02	2	STX	22	34	"	42	66	B	62	98	b
03	3	ETX	23	35	#	43	67	C	63	99	c
04	4	EOT	24	36	$	44	68	D	64	100	d
05	5	ENQ	25	37	%	45	69	E	65	101	e
06	6	ACK	26	38	&	46	70	F	66	102	f
07	7	BEL	27	39	'	47	71	G	67	103	g
08	8	BS	28	40	(	48	72	H	68	104	h
09	9	HT	29	41	)	49	73	I	69	105	i
0A	10	LF	2A	42	*	4A	74	J	6A	106	j
0B	11	VT	2B	43	+	4B	75	K	6B	107	k
0C	12	FF	2C	44	,	4C	76	L	6C	108	l
0D	13	CR	2D	45	-	4D	77	M	6D	109	m
0E	14	SO	2E	46	.	4E	78	N	6E	110	n
0F	15	SI	2F	47	/	4F	79	O	6F	111	o
10	16	DLE	30	48	0	50	80	P	70	112	p
11	17	DC1	31	49	1	51	81	Q	71	113	q
12	18	DC2	32	50	2	52	82	R	72	114	r
13	19	DC3	33	51	3	53	83	S	73	115	s
14	20	DC4	34	52	4	54	84	T	74	116	t
15	21	NAK	35	53	5	55	85	U	75	117	u
16	22	SYN	36	54	6	56	86	V	76	118	v
17	23	ETB	37	55	7	57	87	W	77	119	w
18	24	CAN	38	56	8	58	88	X	78	120	x
19	25	EM	39	57	9	59	89	Y	79	121	y
1A	26	SUB	3A	58	:	5A	90	Z	7A	122	z
1B	27	ESC	3B	59	;	5B	91	[	7B	123	{
1C	28	FS	3C	60	<	5C	92	\	7C	124	|
1D	29	GS	3D	61	=	5D	93	]	7D	125	}
1E	30	RS	3E	62	>	5E	94	^	7E	126	~
1F	31	US	3F	63	?	5F	95	_	7F	127	DEL

Notes

1. DC1 and DC3 are also known as XON and XOFF, respectively.
2. The **Commodore 64** character set ROMs do not support all characters. The following replacements are made:

$5C (92) \ is replaced by £ (Pound sterling currency symbol)
$5F (95) _ is replaced by ← (Left pointing arrow)
$60 (96) ` is replaced by - (Horizontal bar - not the minus sign)
$7B (123) { is replaced by + (Cross - not the plus sign)
$7C (124) | is replaced by ◧ (left half checkerboard)
$7D (125) } is replaced by | (vertical bar)
$7E (126) ~ is replaced by ▒ (checkerboard)

This page is intentionally left blank

APPENDIX B

PROMAL KEY CODES
RETURNED BY FUNCTIONS GETKEY & TESTKEY (HEX)

Table B-1: Commodore 64

Key Legend	Plain	Shift	CTRL	C=	Remarks
<-- (left arrow)	5F	5F	06	5F	(next to 1 key)
1 / !	31	21	90	81	BLK
2 / "	32	22	05	95	WHT
3 / #	33	23	1C	96	RED
4 / $	34	24	9F	97	CYN
5 / %	35	25	9C	98	PUR
6 / &	36	26	1E	99	GRN
7 / '	37	27	1F	9A	BLU
8 / (	38	28	9E	9B	YEL
9 /)	39	29	12	29	RVS ON
0	30	30	92	30	RVS OFF
+	2B	DB	--	A6	
-	2D	DD	--	DC	
British currency £	5C	A9	1C	A8	(pounds ster.)
CLR HOME	13	93	--	93	
INST DEL	14	94	--	94	
Q	71	51	11	AB	
W	77	57	17	B3	
E	65	45	05	B1	
R	72	52	12	B2	
T	74	54	14	A3	
Y	79	59	19	B7	
U	75	55	15	B8	
I	69	49	09	A2	
O	6F	4F	OF	B9	
P	70	50	10	AF	
@	40	BA	--	A4	
*	2A	CO	--	DF	
up arrow	5E	DE	1E	DE	
RESTORE	--	--	--	--	
A	61	41	01	BO	
S	73	53	13	AE	
D	64	44	04	AC	
F	66	46	06	BB	
G	67	47	07	A5	

Table B-1 continued: Commodore 64

Key Legend	Plain	Shift	CTRL	C=	Remarks
H	68	48	08	B4	
J	6A	4A	0A	B5	
K	6B	4B	0B	A1	
L	6C	4C	0C	B6	
: / [	3A	5B	1B	5B	
; /]	3B	5D	1D	5D	
=	3D	3D	1F	3D	
RETURN	0D	8D	--	8D	
Z	7A	5A	1A	AD	
X	78	58	18	BD	
C	63	43	03	BC	
V	76	56	16	BE	
B	62	42	02	BF	
N	6E	4E	0E	AA	
M	6D	4D	0D	A7	
, / <	2C	3C	--	3C	
. / >	2E	3E	--	3E	
/ / ?	2F	3F	--	3F	
CRSR up down	11	91	--	91	
CRSR <= =>	1D	9D	--	9D	
f1 / f2	85	89	--	89	
f3 / f4	86	8A	--	8A	
f5 / f6	87	8B	--	8B	
f7 / f8	88	8C	--	8C	
space	20	20	--	20	
RUN STOP	03	83	*	83	

Notes:

CTRL/RUN STOP aborts program to the PROMAL EXECUTIVE.
SHIFT/C= switches mode (upper & lower <--> upper & graphics) .
Codes shown assume upper & lower case mode.
Alpha-lock (CTRL-A) affects GETKEY but not TESTKEY codes.

Table B-2: Apple II

Key Legend	Plain	Shift	CTRL	Apple	Shift/Apple
ESC	1B	1B	1B	9B	9B
1 / !	31	21	31	B1	A1
2 / @	32	40	32	B2	C0
3 / #	33	23	33	B3	A3
4 / $	34	24	34	B4	A4
5 / %	35	25	35	B5	A5
6 / ^	36	5E	1E	B6	DE
7 / &	37	26	37	B7	A6
8 / *	38	2A	38	B8	AA
9 / (	39	28	39	B9	A8
0 /)	30	29	30	B0	A9
- / _	2D	5F	2D	AD	DF
= / +	3D	2B	3D	BD	AB
DELETE	7F	7F	7F	FF	FF
TAB	09	09	09	89	89
Q	71	51	11	F1	D1
W	77	57	17	F7	D7
E	65	45	05	E5	C5
R	72	52	12	F2	D2
T	74	54	14	F4	D4
Y	79	59	19	F9	D9
U	75	55	15	F5	D5
I	69	49	09	E9	C9
O	6F	4F	0F	EF	CF
P	70	50	10	F0	D0
A	61	41	01	E1	C1
S	73	53	13	F3	D3
D	64	44	04	E4	C4
F	66	46	06	E6	C6
G	67	47	07	E7	C7

Table B-2 Continued - Apple II

Key Legend	Plain	Shift	CTRL	Apple	Shift/Apple
H	68	48	08	E8	C8
J	6A	4A	0A	EA	CA
K	6B	4B	0B	EB	CB
L	6C	4C	0C	EC	CC
; / :	3B	3A	3B	BB	BA
' / "	27	22	27	A7	A2
RETURN	0D	0D	0D	8D	8D
Z	7A	5A	1A	FA	DA
X	78	58	18	F8	D8
C	63	43	03	E3	C3
V	76	56	16	F6	D6
B	62	42	02	E2	C2
N	6E	4E	0E	EE	CE
M	6D	4D	0D	ED	CD
, / <	2C	3C	2C	AC	BC
. / >	2E	3E	2E	AE	BE
/ / ?	2F	3F	2F	AF	BF
` / ~	60	7E	60	E0	FE
<--	08	08	08	88	88
-->	15	15	15	95	95
down arrow	0A	0A	0A	8A	8A
up arrow	0B	0B	0B	8B	8B

NOTES:
Alpha lock (CTRL-A) affects GETKEY but not TESTKEY.

APPENDIX C

ERROR MESSAGES AND MEANINGS

Arranged alphabetically (including punctuation)

*** ERROR: ALREADY LOADED
You have attempted to GET a program which is already in memory. If this is deliberate, UNLOAD the program first and then re-issue the GET command.

*** ERROR: DEVICE NOT READY
Usually this indicates that the disk drive door is not closed. Also check for a disconnected or off-line printer or disk drive, unformatted disk, etc. On the Apple II, this error probably indicates that you changed diskettes without issuing a PREFIX command to select the new volume (PREFIX * can fix this).

*** ERROR: DISK ERROR
This uncommon error message indicates a hardware read or write error on the disk. Check for a disk drive turned off or not connected, etc. It may indicate that your diskette has become damaged or that the disk drive heads need to be cleaned. It may also indicate that part of the operating system has been wiped out in memory by an errant program, or similar difficulties. It may also indicate an attempt to append a write-protected disk or locked file.

*** ERROR: FILE DOES NOT EXIST
You have issued a command to the EXECUTIVE to act on a file which does not exist on the currently selected disk. Remember that the default file extension is ".C". If you are trying to act on a file which does not have a file extension remember to enclose the name in quotes (case sensitive on the Commodore 64). For the Commodore 64 COPY command, if the file is not type SEQ, the type must be specified after a comma inside the quotes, for example "BASICPROG,P". On the Apple II, you may have switched disks without a PREFIX * command.

*** ERROR: FILE ALREADY EXISTS
You have issued an EXECUTIVE command which tried to write a new file with the same name as an existing file. If this was deliberate, DELETE the existing file and try again. Remember that the default file extension is .C. For file names without extensions, the name should be enclosed in quotes.

*** ERROR: ILLEGAL COMMAND SYNTAX
You have issued an EXECUTIVE command with arguments which do not conform to the requirements. Consult the PROMAL USER'S GUIDE for the proper command syntax.

*** ERROR: ILLEGAL FILE/DEVICE NAME

You have issued a command with an illegal file or device name. Check the file naming conventions described in the first part of the PROMAL USER'S MANUAL. For non-conforming files, you must enclose the file names in quotes when using the EXECUTIVE. For the Apple, make sure you are not trying to copy between two disks with the same volume name.

*** ERROR: ILLEGAL OPEN DIRECTION

You have issued a command which tries to output to an input-only device, or visa-versa.

*** ERROR: NOT ENOUGH FREE DISK

You have issued an EXECUTIVE command which has tried to write a file to disk larger than the remaining disk space, or, there are no more free buffers (Apple II) or channels (Commodore 64) available.

*** ERROR: NOT IMP.

The command you have issued is not implemented on your version of PROMAL.

*** ERROR: WRITE PROTECTED

You have issued a command which attempted to write or alter a write-protected disk (or a locked file on the Apple). If you wish, remove the write protect sticker or UNLOCK the file and try again.

*** RUNTIME ERROR: 0 DIVIDE

Division (or %) by zero, or an arithmetic overflow has occurred. If this error occurred during compilation, then a REAL literal number was specified which was out of range. The largest REAL number is approximately 1E+37.

*** RUNTIME ERROR: I-O ILLEGAL DIRECTION

A library routine was called to input or output to a file or device which was opened in a different mode; for example, trying to input from the printer.

*** RUNTIME ERROR: ILLEGAL / UNOPEN FILE HANDLE

A library routine expected to find an open file handle for the first argument, but did not find one. Check for a missing file handle where required. Check to make sure you have properly opened the file or device and have saved the file handle in a WORD type variable. Make sure you have not already closed the handle. An OPEN function call should always be tested for success. If the function returns 0, it was not successful.

*** RUNTIME ERROR: ILLEGAL # ARGS - LIB. CALL

A library PROC or FUNC or machine language routine was called with an invalid number of arguments. Check the LIBRARY MANUAL for the correct arguments required.

*** RUNTIME ERROR: ILLEGAL ARG, LIB. CALL

A library routine was called with an invalid or out-of-range argument. Make sure you are using the appropriate type arguments (e.g., not using a REAL where a WORD is expected). Check the LIBRARY MANUAL for restrictions on arguments.

*** RUNTIME ERROR: ILLEGAL I-O REDIRECTION
An illegal I-O redirection has been made. Only STDIN, STDOUT, and STDJOB may be redirected, and they must be redirected to an open file or device. Check for a missing # in front of STDIN or STDOUT. See REDIRECT in the LIBRARY MANUAL for further information.

*** RUNTIME ERROR: ILLEGAL OPCODE
A program has attempted to execute a non-existent instruction. Make sure you are not trying to use REAL arithmetic after a NOREAL command. It can also be caused by a program destroying itself by writing data into its code space. Check for bad pointers, arrays out of bounds, using a value where an address is required, etc. If this error occurs during compilation, you have attempted to compile a program using REAL data after executing a NOREAL command.

*** RUNTIME ERROR: M/L BREAK
A machine language BRK ($00) instruction has been executed. If this is not expected, it is often a symptom of a piece of a program (probably the PROMAL library) having been zeroed by an errant program. It may also reflect an erroneous definition of an EXTernal routine or failure to load a required software package.

*** RUNTIME ERROR: PROMAL BREAK
A PROMAL PROGRAM (possibly the EDITor or EXECUTIVE) has encountered a $00 instruction at the indicated address. This usually indicates that an program bug has caused part of the program to become zeroed out. Check for bad pointers, arrays out of bounds, using a value where an address is required, etc.

*** RUNTIME ERROR: REQ'D PROGRAM NOT LOADED
A required software package is not in memory. Check to see if you are trying to use REAL arithmetic after a NOREAL command. It may also reflect a defective EXTernal declaration. If this error occurs during compilation, you are trying to compile REAL data after a NOREAL command.

*** RUNTIME ERROR: STACK ERROR
The stack has overflowed. This may indicate that you have a routine which calls itself indefinitely or a recursion error. It may also indicate subroutines nested too deeply or with too many arguments passed. If this error occurred during compilation, you have a statement with an expression which is too complex to compile due to stack limitations (for example, 12 levels of parentheses). This may be aggravated by having many levels of indentation, and by using function calls in the expression. In this case, use intermediate temporary variables for sub-expressions to reduce the complexity of the statement.

ERROR 1:
Illegal character here

The compiler has encountered a character which cannot legally be present at this point in the statement. Check for a missing or extra punctuation mark. If the source file was created by something other than the PROMAL EDITor, check for an embedded tab, linefeed, or other invisible control character (DUMPFILE may help find it).

ERROR 2:
Illegal character constant

A character constant must be a single character enclosed by single quotes (´). Check for missing quote or more than one character. The quote character itself can be written as ´´´´.

ERROR 3:
Illegal string constant

A string constant must be enclosed on both ends by double quotes ("). It may not cross a line boundary. Check for unbalanced quotes. The double quote character can be written inside a string as "".

ERROR 4:
PROGRAM or OVERLAY expected

Your program must start with a PROGRAM statement (or OVERLAY statement). Make sure you are compiling the right file.

ERROR 5:
<Name> expected

The compiler expected to find an identifier (name) at this point in the statement. Make sure you are not trying to use a reserved word as a name.

ERROR 6:
Duplicate name

The identifier has already been declared previously. Make sure you are not trying to define a name that is already defined in the LIBRARY, INCLUDE file, etc. Also make sure that you don´t have a variable that duplicates a procedure, function or data name, or visa-versa. In some circumstances, this error may refer to an identifier on the line above the one shown.

ERROR 7:
= expected

The compiler expected to see an "=" operator at this point in the statement. Make sure that you are not trying to use a simple (un-subscripted) variable as an array, or as a procedure name.

ERROR 8:
Constant expected

The compiler expected to see a constant at this point in the statement. Be sure you are not trying to use a variable name where a constant is required. Remember that you may not have constants of type REAL. Also check to make sure that the value specified for the constant is not out of range (Note: the offending constant may be slightly beyond the row of asterisks on the compiler error message). For example, $100000 is an out of range constant.

ERROR 9:
] expected

The compiler expected to find a right bracket at this point in the statement. Remember that square brackets, not parentheses, are used to delimit PROMAL array subscripts. Check for a missing comma. Remember that you should not specify the size of a DATA or EXTernal array.

ERROR 10:
Illegal data type

The expression does not meet the required data type. Remember that in a DATA declaration defining a string, the type is WORD, not BYTE, because the result is a pointer to the string. Also remember that a FOR-statement index must be a simple variable of type WORD. The choices on a CHOOSE statement must match the type of the expression following the CHOOSE **exactly** (you may need a type cast to make a small numeric constant match a word or integer variable, for example 1:+), and may not be type REAL. Finally, keep in mind that the boolean operators AND, OR, NOT, and XOR operate only on type BYTE.

ERROR 11:
Illegal subscript

Subscripted variables may not be used for local variables or arguments.

ERROR 12:
Variable name expected

The compiler expected to find a variable name at this point in the statement. Make sure that you are not trying to use a reserved word, procedure, function name, or constant for a variable. DATA names may not be the destination for an assignment statement.

ERROR 13:
) expected

The compiler expected a right parentheses at this point in the statement. Check for too many or too few arguments on a function call, or missing or unbalanced parentheses. Also make sure you are not trying to enclose the argument list for a procedure call in parentheses.

ERROR 14:
Illegal expression

The expression does not follow the syntax diagram in Appendix P of the PROMAL LANGUAGE MANUAL. Check for an illegal sequence of operators, missing punctuation, etc. Note that the indirect operators (@<, @+, @-, @.) may not appear after the variable name for an assignment statement (use the global array M instead).

ERROR 15:
is illegal here

The # address operator cannot be used here. The address operator cannot appear on the left hand side of an assignment statement, nor can it be applied to anything except a variable. The # operator must directly precede the variable name.

ERROR 16:
Type name expected

The compiler expected to see BYTE, INT, WORD, or REAL at this point in the statement. The type indicator must precede the variable or function name.

ERROR 17:
BEGIN expected

The statement is illegal at this point in the program. If you have a declaration, check the first word for spelling. If this is an executable statement, you must have a BEGIN statement first.

ERROR 18:
End of line expected

This is a general error message indicating that the compiler could not construct a legal statement with what you have at this point in the line. Check for: too many arguments on a procedure call, subscripts on a simple variable, etc. Also be sure you didn't forget the ";" which must precede a comment.

ERROR 19:
, expected

The compiler expected a comma at this point in the statement. Check for too few subscripts on an array reference, or too few arguments on a function or procedure call.

ERROR 20:
Illegal type name here

The type (BYTE, INT, WORD, or REAL) indicated is inconsistent with prior usage.

ERROR 21:
Not in WHILE or REPEAT loop

The BREAK or NEXT statements may only be used inside a WHILE or REPEAT loop.

ERROR 22:
Statement expected

This is a general error message indicating that the compiler was expecting an executable statement but did not find one. Check for a missing END statement in a prior procedure or function or a misplaced declaration.

ERROR 23:
Wrong # of arguments

The procedure or function call has too many or too few arguments.

ERROR 24:
Indentation error

The statement starts with the wrong indentation. Each level of indentation must be **exactly** two blanks. A statement following a conditional statement must be indented. If this is a statement terminating a conditional block such as an ELSE or UNTIL, it should not be indented as far as the line immediately above it. Each choice of a CHOOSE statement should be at the same level of indentation as the

original CHOOSE keyword; the statements executed for each choice should be indented one level. Check also for a missing ELSE for a choose statement, which is always required.

ERROR 25:
UNTIL expected
A preceding REPEAT statement is not balanced by an UNTIL at the same level of indentation.

ERROR 26:
Unexpected end of file
The compiler reached end-of-file without having reached the END statement in the main program. Check for a missing END statement or INCLUDE statement.

ERROR 27:
Undefined
The identifier indicated has not been previously declared or defined. All variables, constants, procedures and functions must be declared before they are referenced. Check for a missing INCLUDE LIBRARY or other INCLUDE statement, or for a spelling error.

ERROR 28:
Illegal FOR variable
The index variable for a FOR-loop must be a simple (non-subscripted) variable of type WORD.

ERROR 29:
TO expected
The compiler expected the keyword TO to appear at this point in the FOR statement.

ERROR 30:
[expected
The compiler expected to find a left bracket at this point. Remember that square brackets, not parentheses, are used to delimit PROMAL arrays. Make sure you are not trying to use an array name on the left side of an assignment statement without specifying which element of the array should get the result.

ERROR 31:
PROC or FUNC expected
The compiler expected to see the keyword PROC or FUNC at this point in the declaration. This error can also be caused by a missing type indicator (BYTE, INT, or WORD) on an EXTernal variable declaration.

ERROR 32:
AT expected
The COMPILER expected to find the keyword AT at this point in the declaration.

ERROR 33:
Illegal refuge
The keyword REFUGE must be followed by a constant of value 0, 1, or 2.

ERROR 34:
Illegal REAL constant
A literal number is incorrectly formed. Check for the letter O or l instead of zero or one, missing ´.´, etc. On the Commodore 64, check for cross or bar characters instead of + or -.

ERROR 35:
Non-REAL expected
An expression of type REAL is not legal at this point. Subscripts must be type WORD. CHOOSE statements may not have an expression of type REAL. CON declarations may not be REAL (use DATA instead).

ERROR 36:
Illegal import
You have more IMPORT files than are allowed (maximum is 6), or have an erroneous declaration in an imported block.

ERROR 37:
Illegal <import var> :> = ...
You may not use the high-byte operator (:>) on an imported variable appearing on the left hand side of an assignment statement.

ERROR 38:
Illegal export
The declaration cannot be EXPORTed. Only constants, variables, data, procedures and functions can be EXPORTed. You may not EXPORT EXTernal declarations. Check for missing EXPORT keyword on PROGRAM line.

ERROR 39:
Too many dimensions
PROMAL arrays may have a maximum of eight dimensions.

ERROR 40:
Demo compiler can´t IMPORT/EXPORT
The compiler on the PROMAL DEMO diskette does not support EXPORT declarations or INCLUDE files of IMPORTs. You must use the full compiler for these features.

ERROR 129:
Compilation cancelled
This is not an error message, but indicates that compilation was terminated by the operator in response to a prompt.

ERROR 130:
Not enough free memory
The COMPILER cannot find enough free memory for its tables. UNLOAD some programs and try again. On the Commodore-64 using the standard compiler, you will not be able to compile unless the workspace is clear (but you can with the demo compiler). On the Apple II, if you have issued a BUFFERS HIRES, you will need to give a BUFFERS 3 command before compiling.

ERROR 131:
Cannot open object file

The compiler cannot open the object file for writing. Check for a write-protected diskette or full diskette. On the Apple, check for a locked file or diskette change without PREFIX command.

ERROR 132:
Cannot open source file

The compiler could not find the specified source file, or could not successfully open it for reading (for example, drive not ready). Check for spelling errors. The default extension for source files is ".S". On the Apple, check for disk changed without PREFIX * command.

ERROR 133:
Cannot open list file

The compiler could not open the listing file for writing. Check for device not ready, write-protected disk, disk full, etc. The default extension for the list file is ".L".

ERROR 134:
Cannot open export file

The compiler could not open the export file for writing. Make sure that the disk is not write-protected or full (or the file locked on the Apple). The Commodore 64 may not be able to open the export file if you have a listing enabled (due to limitations of the 1541 drive).

ERROR 135:
Cannot open include file

The compiler cannot find or cannot open the specified INCLUDE file for reading. Make sure the desired file is present on the disk. The default file extension is ".S". On the Apple II, this may be caused by not having the proper prefix (volume name). Also, you may have to increase the number of buffers on the Apple if you have a listing file, export file, and/or nested INCLUDE files.

ERROR 136:
No source file, Workspace empty

No source file name was specified on the COMPILE command, and the Workspace is empty. You need to specify a filename to be compiled.

ERROR 137:
Illegal COMPILE argument

The COMPILE command has an illegal argument. See the PROMAL USER'S GUIDE for the correct command syntax.

ERROR 138:
File name duplicates another argument

One of the output file names specified on the COMMAND line is the same as one of the input files, including the extension.

ERROR 139:
Can't write to L device

The L device was specified as an output file for the compiler. This is not permitted.

ERROR 140:
Can't write to Workspace

The W device was specified as an output file for the compiler. This is not permitted.

ERROR 141:
String buffer overflow (Use B option)

The string buffer (literal pool) used by the compiler has overflowed. UNLOAD all programs and use the B option on the COMPILE command. If you have already done this, use the B=Solf option to increase the size of the literal pool (See USER'S GUIDE).

ERROR 142:
Forward Reference overflow

The forward reference table used by the compiler has overflowed. UNLOAD all programs and use the B option on the COMPILE command. If you have already done this, use the B=Solf option to increase the size of the forward reference table.

ERROR 143:
Object buffer overflow (Use B option)

The object buffer used internally by the compiler has overflowed. UNLOAD all programs and use the B option on the COMPILE command.

ERROR 144:
Symbol table overflow (Use B option)

The internal symbol table used by the compiler has overflowed. UNLOAD all programs and use the B option on the COMPILE command. if you have already done this, use the B=Solf option to increase the size of the symbol table.

ERROR 145:
Too many ELSEs

Your program has an IF with more ELSEs than the compiler can handle, or nested loops greater than it can handle.

ERROR 146:
Too many nested loops

Your program has loops nested to a greater depth or complexity than the compiler can handle.

ERROR 147:
INCLUDEs overnested

You have an INCLUDE statement inside an INCLUDE file, which requires opening more files than the Commodore 64 disk or Apple ProDOS will allow.

ERROR 148:
Unbalanced ? (conditional comp.)

You have a ? in column 1 of a statement initiating a conditional block which is not balanced by a matching ? terminating the conditional compilation block. Conditional compilation blocks may not be nested.

ERROR 149:
1st sector rewrite error

For the Commodore 64, this indicates a hardware or firmware disk drive failure or incompatibility. Don't use 2-sided mode on a 1571.

ERROR 150:
B option not in Demo compiler

The B compiler option is not supported in by the PROMAL DEMO COMPILER. You must use the full compiler (on the System Disk) to use the B option.

ERROR 151:
Demo compiler line limit exceeded

The Demo compiler can only compile files with up to 400 lines, excluding comments (but including the LIBRARY). You need to use the full compiler on the PROMAL system disk.

ERROR 152:
Disk write error

The compiler encountered a disk error while writing a file.

ERROR 153:
Disk full

There was not enough room to write the file on the disk.

xxxx ISN'T A TEXT FILE

You have tried to EDIT a file named xxxx which the EDITor thinks is not a text file. Check to see if you are trying to edit a compiled program or data. It may also indicate that the file has lines longer than 125 characters. For the Apple II, it may indicate that the file was prepared with a word processor which sets bit 7 of each character to 1. If this is the case, you can fix the file by using the CLEARBIT7 demo program.

NO BUFFER SPACE

The EDITor could not find enough free space for its buffer. To correct this, type UNLOAD (and WS 0 if you are not using the Workspace on the Commodore 64), and try again.

NOT A PROMAL OBJECT FILE: xxxx

You have attempted to execute a file which is not a compiled PROMAL program or a relocatable machine language program. All PROMAL programs must be successfully compiled before they can be executed. This error usually occurs when you try to execute a program which was compiled using the B option but had compilation errors. It can also occur if you attempt to execute a version 2.0 module on a version 1.X PROMAL system.

NOT ENOUGH FREE MEMORY

The specified program or overlay could not be loaded because there is not enough free memory. If it is a program on the Commodore 64, you will need to set the Workspace size to 0 and try again. If it is an overlay, you need to unload all programs and restart the main program. If the problem persists, your program may simply be too large. Consider modifying it to use overlays, or, use the NOREAL command if

appropriate, to free up more memory. Remember that your variables also require memory; you should consider reducing the size of your arrays. For the Commodore 64, you can gain a lot of space by using a bootstrap loader to set HIFREE and HIMEM to MEMLIM and then loading your program as described in the section on the LOADer.

NOT LOADED OR RELOC ERROR: xxxx

You have attempted to load or execute a program or overlay which imports from the indicated program or overlay, without having that program or overlay loaded first. UNLOAD memory and use the GET command to load any programs needed. You may wish to write a bootstrap program as described in the LOADer section of the PROMAL LANGUAGE MANUAL to load the required modules automatically. If the program name shown is the same as the program you are trying to execute, then there is not enough free memory to relocate your program after loading it. You need to unload other programs or free up additional memory as described above.

OK TO CLEAR WORKSPACE (Y/N)?

This Commodore 64 message is not an error message but a warning. It is given if you specified the B option on the compiler, but there is something in the Workspace. If you reply with a Y, the compiler will clear the workspace and proceed. Otherwise, it will abort.

PROGRAM OR OVERLAY NOT FOUND: xxxx

You have issued an EXECUTIVE command which is not a built in command, nor is it in memory or on disk. Check for spelling errors, and make sure you have the correct diskette. Remember that ".C" will be the default file extension. For the Apple II, you may have changed disks without using the PREFIX command to set the new volume name.

xxxx TOO LARGE TO EDIT

The specified file name, xxxx, is too large to EDIT in the available memory space. To correct this, type UNLOAD to free additional memory and try again. If you are not using the Workspace on the Commodore 64, you should also issue a WS 0 command. If you have already done all this, your file may be too large to EDIT. You can split it into two files using the SPLIT Utility, and then edit each file separately.

USER BREAK

This is not necessarily an indication of an error, but shows that the program was aborted by the user (by CTRL-STOP on the Commodore 64 or by CTRL-C or CTRL-RESET on the Apple).

APPENDIX D

LOCATING RUNTIME ERRORS AND VARIABLES IN MEMORY

The PROMAL nucleus provides runtime checking for many errors such as division by zero, illegal arguments on Library routines, etc. A typical runtime error message would be:

```
*** RUNTIME ERROR: ILLEGAL # ARGS - LIB CALL
AT $72B2
```

This tells you that you attempted to call a Library routine with too many or too few arguments. But where in your program did this occur? The absolute address is given in the error message as $72B2. To find the offending statement in your listing, proceed as follows:

1. Execute a MAP command from the EXECUTIVE.

2. Locate your program's starting address (e.g. "AT 7100") in the MAP display.

3. Subtract this value (using hex arithmetic) from the address displayed with the error message. The result is the relative address from the start of your program, for example $72B2 - $7100 = $01B2.

4. Using your program listing, find the statement (not a variable or data declaration) with an address (shown in the column to the left of the statement) that spans the calculated address. This statement (or possibly, the preceding or next statement) is the one where your error occurred.

You can use a similar technique to DUMP the value of shared global variables and global scalar variables (but not local variables). Shared global variables are arrays of any type, or REAL variables (both simple and arrays). Global scalar variables are non-array BYTE, INT and WORD variables which are not declared inside a PROC or FUNC.

To locate a global scalar variable, add the address shown to the left of the variable's declaration on the listing to the address shown as the starting address for variables in your map display. For example, if your listing shows:

```
8    BYTE MYVAL
```

and the MAP for your loaded program shows:

```
MYPROG     (PRO.)  9/ 3/85 CHKSUM 4B9D
 AT 7100-73FF (VARS: A100-A2FF)
```

then add $08 to $A100, giving $A108. This is the absolute address of your variable, MYAL = $A108. Note that if the variables start at an address above the "SYSTEM SPACE" location on the Apple II, you will not be able to DUMP the correct value of your variable because the EXECUTIVE has re-used that address space.

Locating an array or REAL variable is slightly more complex. Use the SIZE command (or the summary at the end of the listing) to determine the number of bytes of scalar variables used by your program. For example, if the SIZE command displays:

```
MYPROG      (PRO.) 9/ 3/85 VER.2
 CODE $20BA, GLOB VARS $01C0, $09
```

then your program has $09 bytes of scalar variables. Add this number to the address shown in the listing for your variable, and add that result to the starting address of variables shown for the map command. For instance, using the above example, if your listing shows:

```
6A  WORD VALUES [5]
```

then the absolute address of VALUE[0] is at $A173 ($6A + $09 +$A100).

When displaying the value of variables, remember that WORD and INT variables are stored with the low order byte first and the high order byte at the next higher address.

REAL variables occupy 6 bytes each, in the following format:

```
        +-----------------------------------------------------------+
        |S<--Exp--><---------------Mantissa---------------->|
        +-----------------------------------------------------------+
Addr:     +0       +1       +2       +3       +4       +5
```

This format is based on the IEEE standard single precision data format, but is extended with 2 additional bytes in the low order mantissa to increase the accuracy from 6 to 11 significant decimal digits. It uses a binary representation where:

```
S        =  1 bit sign (1=negative value)
Exp      =  8 bit exponent (biased by $7F, 0 if number=0.0)
Mantissa =  39 bit mantissa normalized between 1.0 and 2.0,
            with an implied 1 bit & binary point to left of mantissa.
```

If you are not familiar with floating point representations, you may wish to consult an "elementary" book on computer arithmetic, or the IEEE Floating Point Standard before attempting to interpret the value.

Note for numerical analysts: The PROMAL floating point routines do not support gradual underflow. Any number with an exponent of zero is considered zero. Also, rounding-to-even is not supported (but rounding is). This is of no consequence to normal users.

APPENDIX E

PRINTER SUPPORT

COMMODORE 64 PRINTER SUPPORT

PROMAL supports standard Commodore Printers using the serial interface. All Commodore and Commodore-compatible printers we tested worked without any special effort as the "P" device with PROMAL. Parallel printers using Cardco (Model C through G+) or similar adapters should also function normally.

Printers (or printer interfaces) often have special modes selected on the basis of the "secondary address". The following three variables can be used to control your printer:

```
EXT ASM BYTE C64PSA AT $0DF3  ; Desired secondary address (default 7)
EXT ASM BYTE C64PUL AT $0DF4  ; Bit 7=1=flip case (default=$80=yes)
EXT ASM BYTE C64PDV AT $0DF5  ; C-64 printer device # (default 4)
```

You should set these variables to the desired choices before OPENing the P device. The C64PUL variable controls whether or not lower case and upper case alphabetic characters should be reversed before output to the printer. This is normally needed because PROMAL uses standard ASCII characters but most Commodore-compatible printers expect "Commodore ASCII". If your printer prints alphabetic characters in the wrong case, you can use a **SET DF4 0** command from the EXECUTIVE, or in your BOOTSCRIPT.J file.

The standard Commodore device number for the printer is 4. However, if you have a second printer on the serial bus or are using a plotter, you may wish to open the P device to a different device number. You can do this by installing the desired device number in the byte at $0DF5 (C64PDV).

The Commodore 1525 printer does not support form feeds, so listings will not be properly paginated, but 1526 printers will work properly.

It has been reported that some versions of the Commodore 1526 printer have intermittent problems when used with the 1541 disk drive. These problems are characterized by a serial bus "lockup", which may cause the system to hang inexplicably or to display error number 40 or 41. **This problem has nothing to do with PROMAL** and will also appear with other software. Rather, this is a problem with the Commodore ROMs in the printer and/or disk drive. If you experience these problems, you may want to contact Commodore dealer and request that he upgrade your system to the latest level ROMs. If in doubt, you can find out what ROMs you have in your printer by performing the 1526 self-test. If the prints "CBM COMMODORE 1526/MPM-802 PRINTER - REV 07C", then you have the latest printer ROMs. At the time of publication of this manual, our best information is that the latest ROMs are as follows:

1541 Disk Drive ROM part number 901229-05
1526 Printer ROM part number 325341-08

(Our sincere thanks to Mr. A. Ryan of Ontario who provided this information).

APPLE II PRINTER SUPPORT

PROMAL supports standard Apple printers or compatible printers. For the IIe, the printer card should be installed in slot 1 and conform to the standards for Apple Pascal. For the Apple IIc, the printer should be attached to the printer port (port 1) in the usual fashion.

You can control whether or not PROMAL should automatically send a LF after every CR to your printer by the setting of the following variable:

```
EXT BYTE APLPALF AT $0DF3  ; Bit 7=1=send LF after CR
```

If your printer double spaces when it should single space, set this variable to 0. This can be done from the EXECUTIVE or a JOB file with a **SET DF3 0** command. If it prints one line on top of the other, set it to $80. When outputting graphics or speciual escape sequences, you may need to turn this off (so a $0D graphic data byte won't be interpreted as a CR and cause a suppious $0A linefeed data byte to be sent to the printer).

Also, if your computer is a IIc or is connected by a **serial** interface, you will need to set another variable to 0 to perform graphics or escape sequences. This is not a PROMAL variable, but a global Apple variable that controls the ROM output routines in the Apple:

```
EXT BYTE PRESCCH AT $0638+$C0+1 ; Serial command enable flag, Apple
...
PRESCCH = 0 ; Disable Apple ROM processing of serial printer output
```

This will keep the Apple from processing escape sequences to the serial port internally, and will pass them straight through to the printer.

PROMAL automatically configures an appropriate printer driver for your computer during boot-up. In very rare cases, if you are using a printer card which does not follow the Apple standard, you may have to supply your own printer driver. In this case, See **APPENDIX G**, which tells the location of a pointer to a table of addresses for the printer driver input and output vectors. The table pointed to consists of a WORD holding the address of the initialization entry point, followed by a WORD holding the addres of the output entry point (will be called with character in A).

APPENDIX F

DATA COMMUNICATIONS SUPPORT

PROMAL provides support for serial data communications using RS-232C asynchronous data transmission by the T device. Input and output for EXECUTIVE commands can be redirected to the modem in the same way as to the printer or other device. More frequently, a PROMAL program will perform input and output to the T device. This makes it relatively easy to handle roughly 90 percent of your telecommunications needs. This section assumes you have a basic working knowledge of the fundamentals of data communications, such as baud rate, parity, etc. If you don't, you may wish to consult a reference book, such as RS-232 Made Easy by Martin D. Seyer. You may also need to consult the documentation for your particular modem or RS-232 adapter.

You can specify the baud rate, parity, number of data bits, and number of stop bits desired for the T device before opening it. This can be done either using the TMODE utility program, or by setting values into memory directly from a program.

TMODE UTILITY

The TMODE utility is a PROMAL program provided on disk, which has the following command syntax:

```
TMODE [Baud [Parity [Databits [Stopbits ]]]]
```

If no arguments are given, it displays the current values. **Baud** is the desired baud rate, (default is 300 when PROMAL is started). Legal values can be 110, 300, 600, 1200, 2400, 4800, or 9600. You may also abbreviate 300 as 3, 9600 as 96, etc. When PROMAL is booted up, the initial baud rate is set to 300. The second optional argument is **Parity**, which should be specified as E, O, N, M, or S for even, odd, none, mark or space, respectively. The initial default is none. The third argument is **Databits** which should be 7 or 8. The initial default is 8. The final optional argument is **Stopbits**, which should be 1 or 2. The initial default is 1. Most systems use 1 stop bit except at 110 baud. Optional arguments which are not specified remain unchanged.

The values specified by the TMODE command will take effect the next time the T device is opened (or used in an EXECUTIVE command). The values set by TMODE may also be set directly from a program, discussed below.

PROGRAMMING THE T DEVICE

For many applications, programming the T device is very simple. You just need to open it and then input or output to it the same way you would a file. For example:

```
WORD MODEM  ; File handle for T device
BYTE BUFFER[81] ; Input buffer for T device
...
MODEM=OPEN("T",'B')  ; Open T for input & output
IF MODEM=0  ; Trouble?
  ABORT "#CCan't open T device"
PUTF MODEM, NL,"This is transmitted over the modem.",NL
...
IF GETLF(MODEM,BUFFER,80)
  PUT BUFFER,NL ; Display line received from modem
...
```

The primary added complexity of dealing with a modem is handling the situation where no data is received when it is expected. To handle this, a status routine is provided, to tell you when data is ready to read. In addition, global variables are provided to allow selecting different communications parameters under program control. These definitions are given in the file PROSYS.S and are summarized below:

```
EXT ASM FUNC BYTE GETTST AT $0FC6 ; TRUE if ready. Arg=0 input,1=output.

EXT BYTE TBAUD    AT $0DE9 ; 3=110,6=300,7=600,8=1200,A=2400,C=4800,E=9600
EXT BYTE TPARITY  AT $0DEA ; 0=none, 1=odd, 2=even, 3=mark, 4=space
EXT BYTE TDATAB   AT $0DEB ; 0=8 bits, 1=7, 2=6, 3=5
EXT BYTE TSTOPB   AT $0DEC ; 0=1 stop bite, 1=2 stop bits
EXT BYTE TEOFCH   AT $0DED ; EOF char. (CTRL-Z default), unless TDEVRAW set
EXT BYTE TDEVALF  AT $0DEE ; Auto line feed, $00=no, $80=out,$40=in,$C0=both
EXT BYTE TDEVRAW  AT $0DEF ; "Raw" mode flag, $80 = no EOF or LF processing
EXT BYTE TDEVST   AT $0DF0 ; Status byte from last operation (see below)
```

Function GETTST requires one argument which is either 0 (to test the input status of the T device) or 1 (to test the output status). The function returns TRUE if the serial device is ready and FALSE otherwise. For input, it will return TRUE when at least one character has been received and can be read. For an example of how to use GETTST, see file TINYTERM.S.

The variables TBAUD, TPARITY, TDATAB, and TSTOPB can be used to set the same values which are set or displayed by TMODE, from within a program, for example:

```
TBAUD=$08
TPARITY=2
```

This selects 1200 baud with even parity. The desired values should be set prior to opening the T device. See the SRECEIVE.S and SSEND.S files for examples of how to set these variables.

TEOFCH is used to determine what character should be treated as End-of-File for input from the T device, defaulting to CTRL-Z ($1A). The default value allows a remote serial device to be used for input to the EXECUTIVE by redirecting input to the T device. TEOFCH is particulary significant for programs which use GETBLKF to read from the T device (generally not recommended).

In a program, you will often not want **any** character interpreted as end of file. This can be done by setting the TDEVRAW flag to $80 (not to TRUE!), which causes the T device to pass all characters straight through. The TDEVALF byte is the "auto line feed" flag. Setting bit 7 to 1 causes the T device driver to add a line feed ($0A) automatically after every CR ($0D) is sent. This may be needed it you have a serial printer or another computer connected to the serial port. Setting bit 6 of TDEVALF to 1 causes the driver to discard incoming linefeeds. If the TDEVRAW flag is $80, both TDEVALF and TEOF are ignored.

DETAILED INFORMATION FOR APPLE II T DEVICE

The Apple II T device driver supports the Apple Super Serial card and true compatible cards, and the Apple IIc serial port 2. For maximum flexibility, the PROMAL device driver manipulates the 6551 chip hardware directly. Therefore it does not support the Apple Pascal escape-sequences for selecting communication attributes, etc. (which are unsuitable for many applications). PROMAL does not support buffered T device input using interrupts, because of the incompatibility of some serial cards. This means that your application program may have difficulty "keeping up" with an incoming stream of characters from the T device at higher baud rates if it does extensive screen output or other time-consuming activities. Expert programmers with serious telecommunications applications may wish to write their own interrupt service routine, following the guidelines in the Apple Reference Manual. For this reason, or in order to support incompatible cards, PROMAL leaves "hooks" for writing your own T device drivers. If your serial card does not have a 6551 chip with its data register at $C0A8, you will have to write your own driver to use the T device.

The WORD at $0DF1 (Apple only!) is a pointer to a table of WORDs containing the addresses of the initialization, status, input, and output routines, respectively, used by the PROMAL T device. All are machine language routines. The INIT routine has no arguments and returns nothing. The STATUS routine expects A=0 for input or A=1 for output, returns the status in A, and the carry bit set if ready. The INPUT routine has no arguments and returns the character in A. The OUTPUT routine expects the character in A and returns nothing.

The TDEVST byte is set by any status, input, or output calls, as follows:

Bit 0 = Parity error	Bit 4 = Transmit buffer empty
Bit 1 = Framing error	Bit 5 = DCD not state
Bit 2 = Overrun error	Bit 6 = DSR not state
Bit 3 = Receive buffer full	Bit 7 = Interrupt flag

We have successfully used the T device on the Apple at the full 9600 baud with the built in drivers (for example, the SSEND and SRECEIVE programs). Naturally, this depends on your program. For example, if you attempt to access disk or have another time-consuming activity while characters are received, you will lose characters. In this case you should either arrange to have transmission halted temporarily (for example, using XON-XOFF protocol), or use a machine language buffered interrupt service routine.

DETAILED INFORMATION FOR COMMODORE 64 T DEVICE

The Commodore 64 uses the standard "Kernal" ROM support for RS-232, and is therefore subject to the same limitations. Opening the T device causes a 512 byte buffer to be allocated at LOFREE (an open error may indicate that there is not enough room for this), and LOFREE is moved up accordingly. This buffer is filled and emptied by the non-maskable interrupt routine in ROM. The RS-232 device is always opened in Commodore "3-line" mode; "X-line" is not supported due to problems in the Commodore firmware. For the Commodore, the T device driver will return an end-of-file indication on input if the buffer is empty or the Break detected bit is set in the status.

When using a modem (as opposed to direct connection through an RS-232 level shifter such as the Commodore 1011A), you will need to do additional programming to control the special modem functions. For example, to use the model 1660 300-baud modem, you will need to access the parallel port (user port) to go "off hook" **after** you open the T device. The TINYTERM program illustrates how to do this. For other modems or features such as dialing, you will need to consult your modem manual.

In general, we recommend you do not exceed 600 baud on the Commodore, although we have had success with 1200 baud provided that a long "burst" is not send to the Commodore at the full 120 characters per second. Naturally, this depends on the ability of your program to keep up.

The TDEVST byte reflects the status after any input, output, or status call to the T device, as follows:

Bit 0 = Not functional	Bit 4 = Not functional
Bit 1 = Framing error	Bit 5 = Not functional
Bit 2 = Receiver buffer overrun	Bit 6 = Not functional
Bit 3 = Receiver empty / Transmitter full	Bit 7 = Break detected

SSEND AND SRECEIVE PROGRAMS

The files SSEND.S and SRECEIVE.S on the PROMAL disk are source programs for transmitting files between computers with PROMAL, at speeds of up to 9600 baud on the Apple or 1200 baud on the Commodore 64. The files do not have to be text files; any kind of PROMAL file can be sent.

The programs provided form a complementary pair. SSEND transmits a specified file using an error-correcting protocol, and SRECEIVE receives the file on another computer and installs it on disk. The programs can be used to transfer any size file at up to 9600 baud between computers in close proximity without a modem, by using a simple "null modem" cable between serial ports, as shown below. If used with a Commodore 64, we suggest you limit transmission to 600 baud.

The diagram below illustrates how to wire a direct-connect cable (null modem), with pin connections for the Apple IIc 5 pin connector on port 2, or the Commodore 64 RS-232 adapter model 1011A or similar level shifter.

Computer A					
Apple IIC DIN-5 Pin #	Commodore 64 1011A Pin #	RS-232 signal Name (#)	Computer B RS-232 signal Name (#)	Commodore 64 Port Pin #	Apple IIC DIN-5 Pin #
1	6	DTR (6) -------	DSR (20)	20	5
2	3	TD (3) -------	RD (2)	3	4
3	7	GND (7) -------	GND (7)	7	3
4	2	RD (2) -------	TD (2)	2	2
5	20	DSR (20) -------	DTR (6)	6	1

Two remote computers can also exchange files at the maximum baud rate supported by the modem used (typically 300 or 1200 baud). When using a modem instead of a direct connection, some modifications may need to be made to the program to send modem control commands (such as to answer the phone). These modifications are entirely dependent on the type of modem being used. Consult your modem manual for further information.

To transmit a file, start the **receiving** computer's program first, for example:

```
SRECEIVE MYFILE.T 1200
```

will receive a file called MYFILE.T at 1200 baud. Then start the transmitting computer's program (at the same baud rate, of course):

```
SSEND MYFILE.T 1200
```

The file will be transmitted in 1K blocks with a verification "handshake" after each block is correctly received. If a block is garbled in transmission, the receiving program requests a retransmission of that block. The program exits when the entire file is received.

The source code for SSEND and SRECEIVE has comments which explain the operation of this simple communications protocol in detail. You may freely incorporate any part of these programs in your own projects.

TINYTERM

The TINYTERM program provided in source form on the PROMAL disk is a tiny terminal emulator program which provides the basic functions necessary to access a remote computer using an external modem. This program provides a "bare bones" communication program which can be used to communicate with many remote time sharing services, such as Compuserve. It is not intended to provide the functionality of commercially available communication packages, but is a simple program to illustrate the use of T device. Advanced users may enjoy enhancing it to a full communications package, perhaps adding the ability to upload and download disk files, etc. You may need to modify the program somewhat for use with your modem. We recommend 300 baud operation.

T DEVICE NOTES

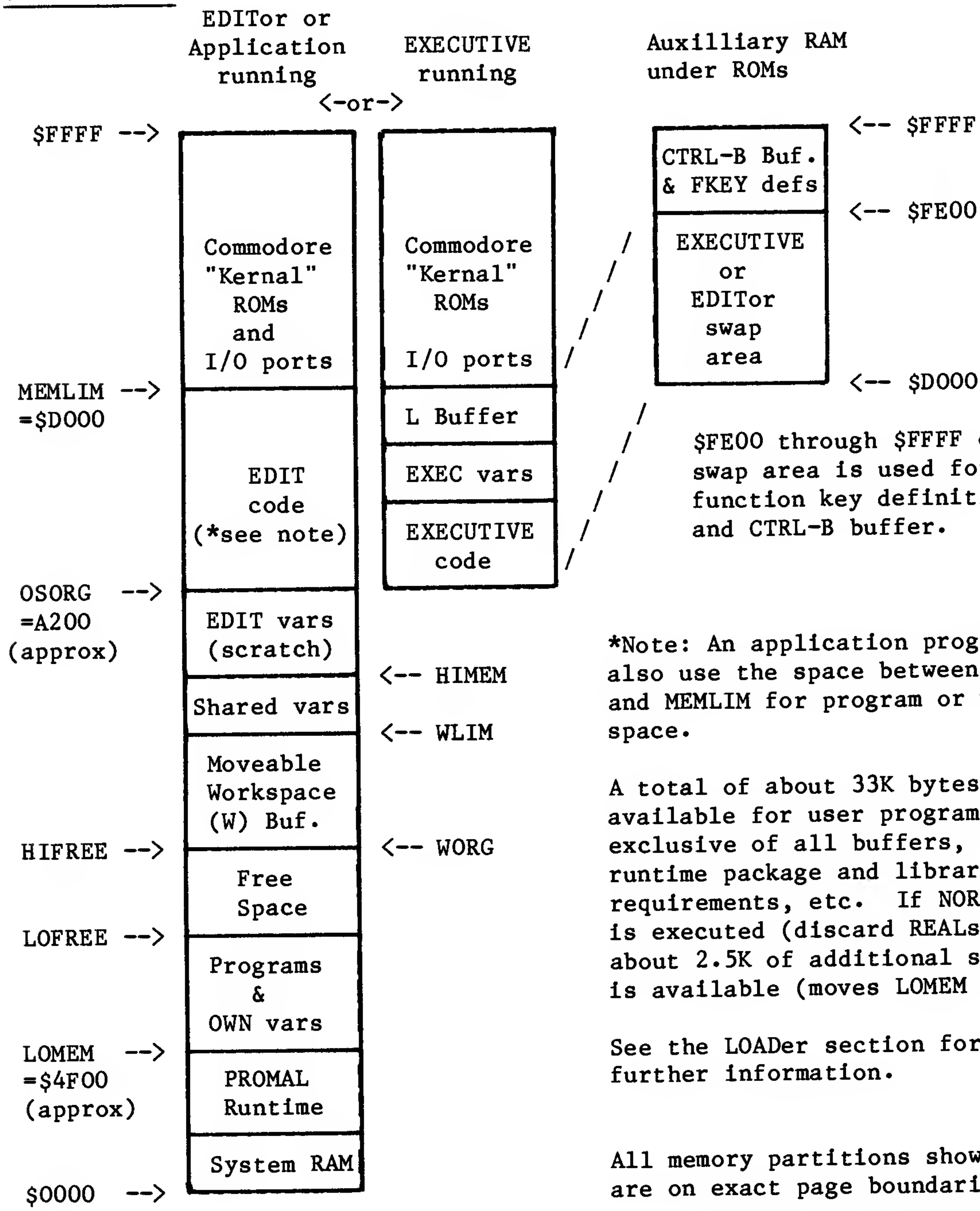
Data communications professionals know that programming serial data transmission is seldom as easy as it looks. It is important to understand that while the RS-232C document defines a "standard" way for computers to communicate, it is an extremely general standard with many possible variations. Not only can parameters such as baud rate and parity be selected in a wide variety of permutations, but there is no standard way for programs to address the modem itself. The port addresses for the modem are different on each computer, and are even different on various serial boards or attachments for the same computer. There is no agreement about what commands should be sent to the modem to make it perform its special functions (such as dialing or going "off hook"). Worse yet, there are a maddening variety of software "protocols" in use which govern the way information should be transferred between devices attached with a serial interface.

All these factors make it impossible to make a "one size fits all" driver for the T device. In implementing the T device, we have tried to make it easy to use for the vast majority of cases, and not impossible for the rest. It is entirely the responsibility of the programmer to insure proper data communications for any particular piece of communications equipment. This should be considered part of the application program.

APPENDIX G

MEMORY MAP

COMMODORE 64



$FE00 through $FFFF of the swap area is used for function key definitions and CTRL-B buffer.

*Note: An application program can also use the space between OSORG and MEMLIM for program or variable space.

A total of about 33K bytes is available for user programs, exclusive of all buffers, runtime package and library requirements, etc. If NOREALS is executed (discard REALs), about 2.5K of additional space is available (moves LOMEM down).

See the LOADer section for further information.

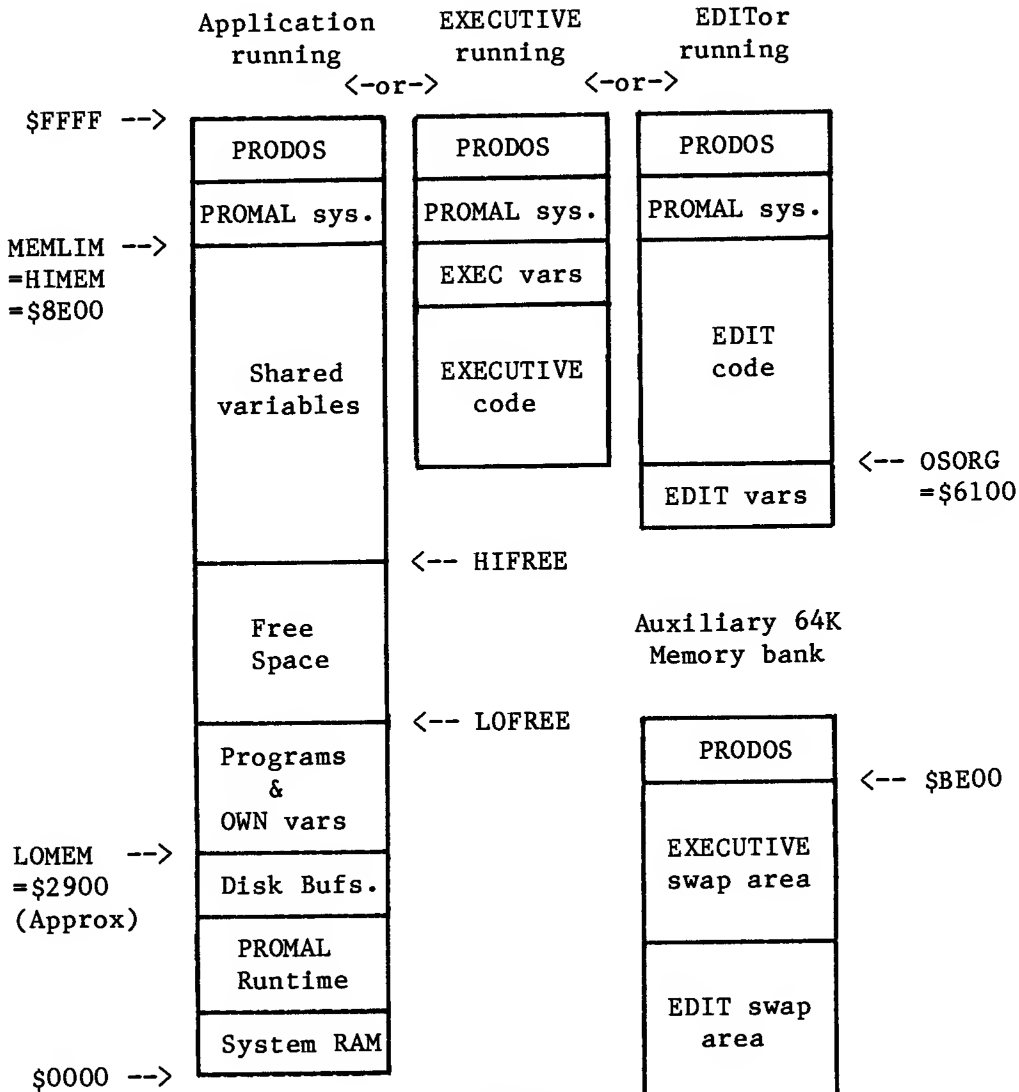
All memory partitions shown here are on exact page boundaries.

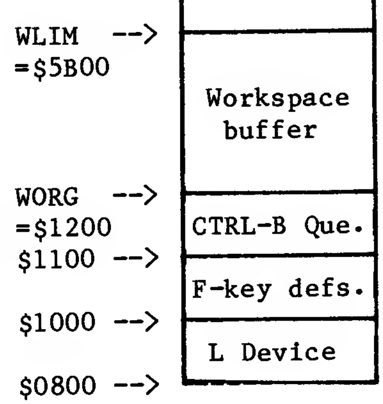
COMMODORE 64

Address	Description
0000 - 0001	6510 On-Chip I-O port
0002 - 0010	Available (Used by BASIC only)
0011 - 0015	Reserved for PROMAL enhancements
0016 - 0019	Used by PROMAL
001A - 002A	Used by C-64 Kernal
002B - 0042	Used by PROMAL
0043 - 0056	Used by C-64 Kernal
0057 - 0089	Used by PROMAL
008A - 00F2	Used by C-64 Kernal
00FB - 00FE	Available
00FF	Used by C-64 Kernal
0100 - 01FF	Hardware Stack
0200 - 0333	Used by C-64 Kernal
0334 - 03FF	Available for M/L programs (see note 2)
0400 - 07E7	Screen memory
07F8 - 07FF	Sprite data pointers
0800 - 08FF	Floating point stack
0900 - 09FF	Heap for Local Variables
0A00 - 0AFF	Scratchpad for I/O, encode/decode, etc.
0B00 - 0DFF	PROMAL System Data Area, see PROSYS.S for details. Reserved.
0E00 - 43FF	PROMAL Vectors, Jump Table, Nucleus, Library, & DYNODISK drivers.
4400 - 4EFF	PROMAL REAL processing routines, **or** Allocatable user memory.
4F00 - A1FF	(Approx.) Allocatable memory for user programs & workspace.
A200 - CFFF	(Approx.) EDITor/EXECUTIVE space, **or** user programs & variables.
D000 - FDFF	(Approx.) EDITor/EXECUTIVE swap area and L device (RAM)
FE00 - FFFF	Function key defs and CTRL-B buffer (RAM)
D000 - FFFF	C-64 Kernal ROMs, VIC, SID, and IO.

Notes:
1. All addresses subject to change without notice.
2. \$0334 - \$036F reserved for PROMAL hi-res graphics package.
3. File PROSYS.S contains definitions of many system locations.
4. See the Chapter 8 of the PROMAL LANGUAGE MANUAL and Appendix H for further information on memory allocation.

APPLE II





Approximately 25K bytes available for user programs, exclusive of all buffers, runtime package, library routines etc. About 2.5K additional can be freed up by executing NOREALS (discards REAL arithmetic), and up to 2K additional can be freed up by selecting fewer than 3 PRODOS disk buffers (BUFFERS command).

APPLE II

Address	Description
0000 - 000E	Available
000F - 0049	Apple II Monitor
004A - 004D	Available
004E - 0055	Apple II Monitor
0056 - 00AF	Used by PROMAL
00B0 - 00FF	Available
0100 - 01FF	Hardware Stack
0200 - 027F	Apple input buffer, used for scratch
0280 - 02BF	Apple input buffer; Scratch path name buffer for PROMAL
02C0 - 02FF	Apple input buffer, used for scratch
0300 - 03EF	Available (See note 2)
03F0 - 03FF	Apple II Vectors
0400 - 07FF	Text and low-resolution graphics display buffer
0800 - 08FF	Floating point stack
0900 - 09FF	Heap for local variables
0A00 - 0AFF	Scratchpad for I/O, encode/decode, etc.
0B00 - 0DFF	PROMAL system data area (see PROSYS.S file). Reserved.
0E00 - 10FF	(approx) PROMAL Vectors, Jump Tables, tables, etc.
1100 - 1BFF	(approx) PROMAL REAL processing, **or** Buffers / allocatable space
1C00 - 1CFF	(can vary) File descriptor table
1D00 - 28FF	(can vary) Disk buffers (3) for ProDOS
2900 - 8BFF	(can vary) Allocatable memory for user programs & variables
6100 - 8BFF	System space for EXECUTIVE & EDIT. Programs may overwrite.
8E00 - BEFF	PROMAL nucleus and library routines.
BF00 - BFFF	ProDOS page
C000 - FFFF	Apple I/O & system memory.

Auxiliary (bank-switched) memory:

0800 - 0FFF	Reserved for L device (library text).
1000 - 10FF	Reserved for function key strings.
1100 - 11FF	Reserved for CTRL-B buffer.
1200 - 5BFF	Workspace buffer.
6000 - BEFF	Swap area for EDIT and EXECUTIVE.

Notes:
1. All addresses subject to change without notice.
2. \$0334 - \$036F reserved for PROMAL hi-res graphics package.
3. File PROSYS.S contains many definitions of system locations.
4. NOREAL command causes allocatable space to start at 1E00 normally.
5. BUFFERS command can change allocatable space.
6. Developer´s version allows applications without auxiliary (bank switched) memory or 80 column card to run (see Developer´s guide).

IMPORTANT SYSTEM DATA AREA ADDRESSES

The following global variables are more precisely defined in the file PROSYS.S unless otherwise noted.

Address	Description
0BB0-0BBB	LDNAME - Command name of last LOAD attempt
0BC0	LDNOCHK - Flag, $80 if bypassing checksum check during loading
0C00-0C03	STDIN, STDOUT File handles (defined in LIBRARY.S file)
0C08	IOERROR - Error code from disk functions (LIBRARY.S)
0C0B	BFILTYP - System-dependent file type for OPEN.
0C0C	DIOERR - Disk I/O error, 0=ok,1=full,2=read err,3=wrt err.
0C0D	DFEXT - ´C´, default file extension for PROMAL files.
0C12-0C15	DATE - Day, Month, Year-1900, 1 byte each
0C16-0C17	LOMEM - Start of Allocatable Memory
0C18-0C19	LOFREE - Next available address for program load, $XX00
0C1A-0C1B	HIFREE - First address not allocatable for programs, $XX00
0C1C-0C1D	HIMEM - End of normally allocatable memory + 1, $XX00
0C1E	LDERR - Loader error return code, $00=success
0C1F	NLT - Number of loaded modules, including EDITor, EXEC.
0C22-0C23	RANDWD - Seed for random number generator (non-zero)
0C2B-0C2C	OSORG - Starting address for EDITor/EXECUTIVE, $XX00
0C2D-0C2E	MEMLIM - End of usable memory (if EDITOR discarded, C-64).
0C51-0C52	MLP - Address of subroutine called by PROC JSR
0C53-0C57	REGA, REGX, etc. - Registers for GO command or BRK
0CF2	NOFNCHK - Flag, if TRUE defeat default file extension
0CFF	BLINKD - Blink delay for cursor. >$7f=solid, 0=invisible.
0D00-0D50	CLINE - Current Command line, complete (LIBRARY.S)
0D51	NCARG - Number of arguments passed on command line (LIBRARY.S)
0D52-0D73	CARG - Array of pointers to arguments on comd. line (LIBRARY.S)
0D74-0DC4	COMD - Command line split into argument strings
0DC5-0DCC	WORG, WPTR, WEOF, WLIM - Pointers for Workspace (see also 0DDB)
0DCD-0DCE	LORG, LPTR, LEOF, LLIM - Pointers for Library (under ROMs, C-64)
0DDB-0DDC	WSIZE - Current Workspace size
0DDD-0DDE	GVORG - Address of start of all shared variables, $XX00
0DE9	TBAUD - T baud (3=110,6=300,7=600,8=1200,$A=2400,$C=4800,$E=9600)
0DEA	TPARITY - T device parity (0=none,1=odd,2=even,3=mark,4=space)
0DEB	TDATAB - T device data bits (0=8,1=7,2=6,3=5)
0DEC	TSTOPB - T device stop bits (0=2,1=2)
0DED	TEOFCH - T device end-of-file char for input (default=CTRL-Z)
0DEE	TDEVALF - T linefeed, bit 7=1=add on output, bit 6=1=strip on input
0DEF	TDEVRAW - T Raw mode flag, $80 = pass all chars through as is
0DF0	TDEVST - T device status for last operation, system dependent
0DF6	DRTERR - Copy of runtime error (See Developer´s Guide)
0DFD	PBLKCNT - # bytes actually written on last PUTBLKF
0DFF	ALPHALK - Keyboard Alpha lock flag, $80 = upper case only
1000	BKEYDEL - Key for delete with pullback
1001	BKEYINS - Key for begin insert mode
1002	BKEYJS - Key for jump to first char of line
1003	BKEYJE - Key for jump to last char of line
1004	BKEYCEL - Key for clear to end of line
1005	BKEYALK - Key for alpha lock toggle

```
1006        BKEYCAN - Key for cancel line
1007        BKEYBT  - Key for backtrack prior line
1008        BKEYBS  - Key for backspace
1009        BKEYTAB - Key for tab (indent)
100A        BKEYRT  - Key for cursor right
100B        BKEYLFT - Key for cursor left
100C        BKEYFK1 - Key for first function key
100D        BKEYFKL - Key for last function key
100E        BKEYEOF - Key for E-O-F from keyboard
```

Note: All addresses subject to change without notice.

System addresses for **Commodore 64** only:

```
0C68-0C6A   PREFIX   - Current drive prefix string
0CFB        EDRES    - Flag, $80 if EDITOR is in memory, $00 if not
0DE0        C64DDV0  - C-64 disk device # for logical drive 0 (default=8)
0DE1        C64DDV1  - C-64 device # for drive 1 (9 for 1541, 8 for MSD)
0DE2        C64N1541 - Flag, $80 = permanently disable DYNODISK
0DE3        C64DYNO  - Flag, $80 = DYNO on , $00 = DYNODISK off
0DF3        C64PSA   - Secondary Address for Printer OPEN (See Appendix D)
0DF4        C64PUL   - Printer upper/lower case switch flag (See Appendix D)
0DF5        C64PDV   - Device number for printer (See Appendix D)
```

System addresses for **Apple II** only:

```
0C68-0CA3   PREFIX   - Current volume & pathname (ends with '/')
0DE0        ABORTCH  - Program abort character (default=CTRL-C, $00=none)
0DE5        RAMUNIT  - /RAM unit number, normally $B0=slot 3 drive 2
0DE6        DSLOT    - Slot for 1:, 2: drive, normally 6
0DF1-0DF2   TDEVTBL  - Address of T device driver table (see Appendix F)
0DF3        APLPALF  - Auto line feed flag for printer ($80=yes, $00=no)
0DF4-0DF5   APLPJT   - Pointer to printer driver vector table (points
                       to Init ptr word, Output ptr word (A=char)
0DFB-0DFE   DSKBUFS  - Pointer to disk buffers.
```

Notes:

1. All addresses subject to change without notice.
2. File PROSYS.S contains the definitions of these and other system locations.

APPENDIX H

DYNAMIC MEMORY ALLOCATION

The PROMAL COMPILER and EDITOR use un-allocated memory for scratch buffer space, and under some circumstances use the Workspace and (in the case of the COMPILER "B" option on the Commodore 64) the space normally occupied by the EDITOR for buffer space. With care, user-written programs may also do this.

Refer to the Memory Map of the System Area in **Appendix G** to understand the variables referred to in this discussion, which are defined in file PROSYS.S. Additional information is contained in the section on the LOADer.

When a user program begins execution, it can safely use all memory between LOFREE and HIFREE for buffer space. This is the area shown as "FREE SPACE" by the MAP command.

On the Commodore 64, you may also safely use the Workspace for a buffer if it is empty. The Workspace exists between WORG and WLIM-1. If WEOF = WORG, then the Workspace is empty. In any event, the space between WEOF and WLIM-1 is unused (this is the "FREE WORKSPACE" area shown by the MAP command). Naturally these pointers will change if you write the Workspace in your program. They may also move if you use the LOADer to load a program or overlay which does not OWN its variables (not recommended). You can "force" the Workspace clear by setting WEOF and WPTR to WORG.

On the Commodore 64, if you want to use the space occupied by the EDITOR for a buffer, you may do so. This is the space between HIMEM and MEMLIM. If you specify "OWN" on the PROGRAM line of your program, for example,

```
PROGRAM MYPROG OWN
```

then you may use memory between WLIM and MEMLIM. This is because specifying "OWN" on the PROGRAM statement forces the PROMAL EXECUTIVE to allocate your global variables at the end of your program rather than at the high end of memory as it normally would. If using the EDITor space, you should set the EDRES flag to 0 (defined in PROSYS.S). The EDITor will be reloaded from disk if needed later.

Note: This applies **only** while in a user-program. This area of memory is absolutely vital to the EXECUTIVE when it is running.

By using the NOREAL command and using all the above techniques, it is possible to free up more than 34K bytes of contiguous space for a user program and data on a Commodore 64.

On the Apple II, about 28K can be made available for programs and variables by using a NOREAL command.

This page is intentionally left blank

APPENDIX I

CALLING MACHINE LANGUAGE ROUTINES FROM PROMAL

This Appendix describes how to call machine language subroutines from a PROMAL program. If you are not familiar with 6502 machine language programming, you may want to skip this section. Because PROMAL is functionally very close to machine language, it is normally not necessary to use any machine language programming at all with PROMAL. However, if you want to use machine language routines, a clean interface is provided. You can even pass arguments to a machine language routine, just like a PROMAL subroutine.

The way you call machine language routines depends on what you want to do. We might categorize the usual needs as follows, in order of increasing complexity:

1. Call a ROM routine that is built in to your computer.
2. Call a small routine you wish to embed in DATA statements as part of your PROMAL program.
3. Call a separate subroutine package, possibly with many routines and passed arguments.

These cases are well-supported with PROMAL. We will address each in order.

PROMAL has a very powerful way of calling machine language routines. It is especially useful for calling ROM-resident routines, such as the **Commodore Kernal** routines or **Apple II Monitor**. Virtually any 6502 machine language subroutine can be called directly from PROMAL with this method. This includes subroutines which expect arguments passed in registers, or return values to the caller in registers. This method can also be used to call machine language subroutines embedded in PROMAL DATA statements. The key to this extremely powerful and simple capability is the built-in JSR procedure, described below.

DECLARATION: **EXT ASM PROC JSR AT $0FB4** (defined in PROSYS.S)

USAGE: **JSR** [Address [,Areg [,Xreg [,Yreg [,Flags]]]]]

Procedure JSR calls a machine language subroutine at a specified address, optionally loading the 6502 processor's hardware registers with specified values before the call. **Address** is the address of the desired routine. **Areg**, **Xreg**, **Yreg**, and **Flags** are optional arguments which specify the desired values to be installed in the A, X, Y, and flags (processor status word) registers, respectively. All register arguments should be type BYTE. Naturally the address must be type WORD.

In order to use the JSR procedure, you will want to include the following declarations in your PROMAL source program (or **INCLUDE PROSYS**, since PROSYS.S contains all these definitions):

```
EXT ASM PROC JSR AT $0FB4  ; entry point
EXT WORD MLP  AT $0C51     ; Subrt. addr.
EXT BYTE REGA AT $0C53     ; A
EXT BYTE REGX AT $0C54     ; X
EXT BYTE REGY AT $0C55     ; Y
EXT BYTE REGF AT $0C56     ; Flags
```

These lines declare the location of the built-in procedure JSR, and the memory locations of copies of the processor register contents to be used. These will be explained presently.

Here is how JSR works. When your PROMAL program executes a JSR statement, the Address argument is copied into MLP, and any additional arguments are copied into REGA, REGX, REGY, and REGF, in that order. The 6502 registers are then loaded as follows:

REGA into the A register
REGX into the X register
REGY into the Y register
REGF into the flags (processor status word)

and a machine language jump to subroutine is performed to the address in MLP.

When the called machine language program returns (with an RTS instruction), the contents of the registers will be saved in REGA, REGX, REGY, and REGF before resuming execution of the next PROMAL statement. Your program can therefore examine the contents of the registers at the time of return. This is important since many machine language routines return values in the registers.

Any optional arguments on your JSR statement which are not specified are not changed. Therefore, for instance, if you JSR to one machine language routine and then JSR to a second routine with no registers specified, the registers will contain the values returned by the first routine.

Some examples should illustrate the simplicity of this method. The examples below refer to Commodore 64 "Kernal" machine language routines, as defined in The Commodore 64 Programmer's Reference Manual. All the examples assume you have added the declaration lines given above.

EXAMPLE 1:

```
    ; Call SCREEN Kernal routine - returns X=columns, Y=rows...
    JSR $FFED ; Kernal "SCREEN" routine
    OUTPUT "#C SCREEN IS #W COLUMNS BY #W ROWS",REGX,REGY
```

This program fragment calls the machine language routine at $FFED without specifying any registers. It then prints the contents of the X and Y registers which were returned by the subroutine.

EXAMPLE 2:

```
CON CHKIN = $FFC6   ; Kernal
CON CHRIN = $FFCF   ; I/O
CON CLRCHN = $FFCC  ; routines
BYTE LINE[81]
WORD I
BEGIN
JSR CHKIN, 0,15     ; channel 15
I=0
REPEAT
  JSR CHRIN         ; input char
  LINE[I]=REGA      ; install char
  I=I+1             ; next location
UNTIL REGA=CR       ; end of line?
LINE[I-1]=0         ; replace CR with end of line
JSR CLRCHN          ; release channel
PUT NL,LINE,NL
END
```

This program fragment reads a line from the disk error channel and prints it on the screen. It will normally display:

```
00, OK,00,00
```

The JSR CHKIN calls the Kernal CHKIN routine, passing 0 in the A register and 15 in the X register to select channel 15. The loop repeatedly calls CHRIN, which returns the character read from the disk drive in register A. The value returned in register A is then installed in a string. When a CR is received from the drive, the channel is closed, the string terminator added, and the line displayed using an ordinary PUT call. Note that when using Commodore channels like this you should be careful not to mix normal PROMAL I/O calls at the same time your channel is open, because PROMAL uses Commodore channels to do its I/O, and only one channel can be selected at a time. Also, channel 15 is always open to the Command/Error channel of the disk drive. If you open your own channels, you should use channels and secondary addresses of 8 or 9 to avoid conflicts with normal PROMAL disk files. Be sure to close them, too!

NOTES ON JSR USAGE

1. When you call JSR, the address is also an optional argument, although it is usually specified. If no arguments are specified, a call is made to whatever address is in MLP. In this way, you can call one out of a number of possible subroutines selected from a table, by putting the desired address in MLP before each JSR call.

2. If a machine language routine executes a BRK (breakpoint) instruction, the address of the breakpoint and registers are stored in these same locations before control is returned to the Executive.

3. For the Commodore 64, Do not attempt to call any routines in the BASIC ROM because PROMAL switches the BASIC ROM out of the Commodore 64 memory map.

4. For the Commodore 64, the file REL_FILES.S on the PROMAL System disk and DISKETTE.S on the optional Developer's disk contain many illustrations of how to use JSR to perform I/O using the Kernal. For the Apple, PRODOSCALLS.S illustrates several examples of JSR usage.

5. The REGF register contains a copy of the processor flags. You can test the flags returned by using the following statements:

```
IF REGF AND $01     ; true if the Carry flag is set
IF REGF AND $02     ; true if the Zero flag is set
IF REGF AND $80     ; true if the Minus flag is set
```

CALLING A MACHINE LANGUAGE ROUTINE EMBEDDED IN DATA STATEMENTS

Many BASIC programs have machine language subroutines embedded in DATA statements. These instructions are READ and POKEd into some unused area of memory and then executed with a USR or SYS statement. You can also embed a machine language routine (or routines) in PROMAL DATA statements, and execute the code using JSR. It is quite a bit simpler than BASIC though, because you do not need to use a loop to READ it and POKE it first.

There are two ways to set up an embedded machine language routine, depending on whether your routine is address-dependent or address-independent. An **address independent** routine is one which will execute correctly regardless of the address at which it is loaded. An address-dependent routine will only execute properly at the address for which it was assembled. This distinction is important because, in general, your compiled PROMAL program (and therefore your data statements) will not be loaded into memory at the same location every time.

If your routine is address-independent (runs anywhere), then you can execute your machine language routine by simply using procedure JSR to call it by name. If your program is address-dependent, then you will have to insure that it is executed every time in a known location. The easiest way to do this is to use procedure BLKMOV to copy it to a known location and then JSR to this known location. This method is equivalent to the READ-and-POKE loop method used in BASIC.

A machine language routine will be address-dependent if it contains any references to addresses within the routine itself. For example, if your routine does JMPs or JSRs to labels that are part of the routine itself, it will not be address-independent. The same applies for a LDA of any data in the routine. Conditional branches are okay, though, because they are coded as displacements, not absolute addresses.

EXAMPLE:

```
BYTE LINE[81]
WORD I

DATA BYTE TOLOWER [] =
$C9,'A',       ;  TOLOWER CMP #'A'
$90,6,         ;          BCC SKIP
$C9,$5B,       ;          CMP #'Z'+1
$B0,2,         ;          BCS SKIP
$69,$20,       ;          ADC #'a'-'A'
$60            ;  SKIP    RTS

BEGIN
PUT "ENTER A LINE: "
GETL LINE
I=0
PUT NL,"IN LOWERCASE ONLY = "
WHILE LINE[I]           ; not end of string?
  JSR TOLOWER, LINE[I]  ; convert char
  I=I+1                 ; bump pointer to next char
  PUT REGA              ; show returned result
PUT NL
END
```

The program fragment above illustrates a call to an address-independent machine language subroutine embedded in data statements. For simplicity, a trivial routine was selected, which simply converts a character passed in the A register to lower case if it is upper case and returns it in A. The line,

```
JSR TOLOWER, LINE[I]
```

calls the embedded machine language routine, no matter where the program is loaded, passing the character desired in the A register. Of course, the actual conversion to lower case could be done much simpler with the PROMAL statements,

```
IF LINE[I] >= 'A' AND LINE[I] <= 'Z'
  LINE[I] = LINE[I] + $20
```

but this is, as we said, simply for illustration.

If your machine language routine is address-dependent, you will need to copy it to some unused memory area and then execute it, for example:

```
...
CON MYSUB = $0334          ; Where to put M/L sub
DATA BYTE MYSUBCODE [] =
... ; (put hex code for routine here)
DATA BYTE SUBEND [] = 0      ; dummy byte to compute loc. of end of code
...
BLKMOV MYSUBCODE,MYSUB,SUBEND-MYSUBCODE ; Copy routine to known loc.
...
JSR MYSUB
...
```


In this example you should assemble your routine for a starting address of $0334, of course.

WRITING MACHINE LANGUAGE EXTERNAL PROCEDURES AND FUNCTIONS

For medium or large assembly language packages, embedding machine language programs in data statements is not practical. For this situation, there are two more ways to interface PROMAL to your assembly-language routines. Both of these methods involve writing a separate assembler program and assembling it. The resulting machine language program is then loaded from disk by your PROMAL program and executed when needed. Your assembly package can have any number of subroutines, which may be either procedures or functions, and are called by name, just like a PROMAL routine.

There are two ways your machine language routine can be loaded into memory. The simpler but less powerful way is to use function MLGET, described below, to load your program at a specified address in memory. MLGET can load machine language programs generated by virtually any assembler for your computer. The only trouble is, you have to find a place to put your program. Since PROMAL allocates memory dynamically for programs, you will have to choose carefully to avoid assembling your program for a location which may be occupied by some other program. There are a few "holes" in the memory map, discussed below, where you can locate your machine language routine using this method. However, if your program is large, you should probably not use MLGET to load your program.

The second method is extremely powerful. This is to create a relocatable machine language PROMAL module, which can be executed by simply typing its name from the EXECUTIVE, or can be loaded under program control with the LOAD procedure. PROMAL 2.0 provides a utility program on the PROMAL diskette called RELOCATE, which has the ability to turn any assembly program into a relocatable program. Your program does not have to be address-independent. It can be used with virtually any assembler. If you don't have an assembler, one is provided that runs under PROMAL, on Volume 1 of the PROMAL Public Domain User Library (available from SMA). By using RELOCATE, you can have a machine language module which will run properly at any address which the PROMAL loader can find available for it. This technique is described at the end of this section.

NON-RELOCATABLE MACHINE LANGUAGE ROUTINES USING MLGET

The biggest problem of a standard machine language routine is where to put it. As you know, PROMAL programs are relocatable, and the EXECUTIVE automatically finds a spot for them in memory. Unfortunately, 6502 machine language programs are not generally relocatable, and will only work properly at the address they were assembled for. Although the PROMAL EXECUTIVE can load a non-relocatable machine language program into memory with the GET command, it won't keep track of where it is, and may allocate a PROMAL program (or variables) right over the top of it if care is not taken.

If your machine language routine is short, one best place to put it is at $0334 to $03FF. This area is available on both the Apple II and Commodore 64. However, the optional PROMAL HIRES GRAPHICS PACKAGE uses this area for global variables, so you should avoid using this area if you will be using hi-res graphics in your application. If your routine(s) take more than 200 bytes, you'll have to find another spot. If you are **certain** your program won't need

to do REAL operations, you can use the 256 bytes at $0800 for your machine language program (this area is used for allocating local REAL variables and performing REAL arithmetic).

For large machine language routines, you may want to pick a spot in the "unused" area shown by the EXECUTIVE MAP command. Be aware that this area expands and contracts as programs are loaded or unloaded, and that the EDITOR or COMPILER will use this area for buffer space. Therefore you will have to reload your machine language code from disk after you use the EDITOR or COMPILER. PROMAL allocates programs from the bottom of available memory up, and allocates global variables (for arrays and global REAL variables) from the top of available memory down. To find a safe spot, first UNLOAD any un-needed programs. Then GET your PROMAL program into memory. Then use the MAP command to determine where the "available space" starts. Round this address up to a nice round number to leave room for future growth of your program, and use this for the address of your machine language program segment. For example, if the free space goes from $5F00 to $7A00, you might want to pick $6000 as the starting address of your M/L code. **Appendix G** gives a PROMAL memory map.

Once you have decided where to assemble your program, the next problem is where to put your "zero page" variables. The only zero page locations you can use with complete safety are:

Available zero page for Commodore 64

```
$02 - $10 (not used by PROMAL, but used by BASIC)
$FB - $FE (the same space that is free for BASIC)
```

Available zero page for Apple II

```
$00 - $0E, $4A- $4D, $B0 - $FF
```

As you already know if you've done much 6502 machine language programming, the Commodore 64 system software uses up almost all of page 0. Unfortunately, this situation is not greatly improved with PROMAL.However, if you just need some scratch space for pointers and the like, you can use the following locations:

Scratch zero page locations for Commodore 64

```
$16 - $19   ; Used for scratch by PROMAL
$36 - $41   ; Used for scratch by the LIBRARY routines
$57 - $66   ; Used only for REAL arithmetic - free if no REALs needed
```

Once you have settled on where to put your program and zero-page variables, the hard part is over. Calling your machine language routine from a PROMAL program is very easy. All you have to do is declare the name of the routine and where its entry point is, for example:

```
EXT ASM PROC MYROUTINE AT $0334
EXT ASM FUNC BYTE TESTIT AT $0337
```

These declarations define two external assembly language (ASM) routines located at $0334 and $0337. It is not necessary to define what arguments (if any) will be passed to these routines. The compiler will accept any number of arguments when calling an EXT ASM routine.

PROMAL calls EXT ASM routines with a 6502 JSR instruction. If your routine is declared as a PROC and doesn't require any arguments, you can simply write it like any 6502 subroutine and just return when you are done via an RTS. More often, though, you will want to receive one or more arguments from the calling PROMAL routine.

PROMAL passes arguments on the hardware stack. All arguments are passed as 2-byte quantities, even if the argument evaluates as type BYTE (the high order byte will be 0 in this case). Passing REAL arguments to assembly language routines is not recommended. On entry to your routine, the Y register will contain the number of arguments passed on the stack. These arguments were pushed on the stack before the JSR, so they are logically "underneath" the return address. Generally you will want to pop off the return address and save it, then pull off the arguments (the last argument will be popped first) and save them in variables of your own. When the routine is done, you should push the saved return address back on the stack and return. You don't have to preserve any registers.

The following example shows how to write an assembly-language procedure with one argument expected to be passed from the PROMAL calling program:

```
;   Sample assembly language procedure MYPROC with 1 argument...

         *=$0334             ; in unused piece of memory...
MYPROC    PLA
          STA  RA            ; save return addr. low...
          PLA
          STA  RA+1          ; & hi byte
          PLA
          STA  ARG1+1        ; save passed argument hi
          PLA
          STA  ARG1          ; & low byte
          ...
;   Operate on ARG1 as desired here...then...
          ...
          LDA  RA+1
          PHA                ; put return addr back on stack
          LDA  RA
          PHA
          RTS                ; return to caller
RA        *=*+2              ; save for return address
ARG1      *=*=2
          .END
```

Here is the companion PROMAL declaration and a sample call:

```
EXT ASM PROC MYPROC AT $0334
...
MYPROC X+1    ; pass X+1 to m/l routine
...
```

If your machine language subroutine is to be a function, it should return its value on the top of the stack. If it is type BYTE, it should only return a byte on the stack, otherwise it should return two bytes.

To assemble your routine you can use any Assembler or Machine Language MONITOR which produces a standard Commodore machine language PRG file or Apple II BSAVE type file respectively as output. An assembler which runs under PROMAL is available in the PROMAL public domain library. Some of the small Machine Language Monitors such as the version of C64MON which loads at $8000 can be run directly from PROMAL for the Commodore 64. Others will have to be run from BASIC. Once you have saved your object file on disk, you can load it into memory from the PROMAL EXECUTIVE with the GET command or by using the MLGET function. When using GET, enclose the name of the file in quotes to indicate that it is a machine language file instead of a PROMAL program. Also be careful to type the name exactly as it is stored in the directory (usually with upper case letters). For example:

```
GET "MYPROG"
```

will load the machine language file "MYPROG" into memory at whatever address it was saved. Note that the MAP command will not show the location where this program is loaded. Alternatively, your application can load the machine language file itself, using the built-in function MLGET, described in below. This is the preferred method.

Let us now look at a slightly more complex example. This example illustrates a machine language function with one required argument of type WORD and one optional argument of type BYTE, defaulting to ´ ´ if not specified:

```
;   Assembly Language function MYFUNC (WORD [,BYTE])
         *=$0334
MYFUNC   PLA
         STA  RA            ; save return address
         PLA
         STA  RA+1
         LDA  #´ ´          ; default if only 1 arg specified
         CPY  #2
         BNE  MYFUNC2       ; branch if only 1 arg specified
         PLA                ; else discard dummy hi byte
         PLA                ; get byte argument specified
MYFUNC2  STA  ARG2          ; save default or specified 2nd arg
         PLA
         STA  ARG1+1        ; save hi byte of arg 1
         PLA
         STA  ARG1          ; save low byte of arg 1
         ...
;   operate on arguments as desired here...then...
         PHA                ; push result to be returned to caller
         LDA  RA+1
         PHA                ; push return address
         LDA  RA
         PHA
         RTS                ; return to caller
```

Here is the companion PROMAL declaration and sample calls:

```
EXT ASM FUNC BYTE MYFUNC AT $0334
WORD WHERE
BYTE CHAR
...
CHAR = MYFUNC(WHERE)          ; call with default for 2nd arg
...
IF MYFUNC(WHERE-1,'A')        ; call with 2nd arg specified
  ...
```

CALLING LIBRARY ROUTINES FROM MACHINE LANGUAGE PROGRAMS

You may call LIBRARY routines from your machine-language subroutines (but you may not call subroutines written in PROMAL). The address of the desired routine can be obtained from the listing of the LIBRARY.S file in Appendix **Q** of this manual. Pass your arguments on the stack, remembering that passed arguments are always 2 bytes each. **Don't forget to set the Y register to the number of arguments you are passing.** The following example shows how to print a message on the screen from an assembly language routine by calling a library routine:

```
PUT      =    $0F15          ; Address of PUT routine (from Library)
         ...
;   Print an error message and then the character now in the X reg, then CR.

         LDA  #<ERRMSG
         PHA                 ; Push the address of the string to print (low)
         LDA  #>ERRMSG
         PHA                 ; then hi byte
         TXA                 ; push the character to print after the msg.
         PHA
         LDA  #0
         PHA                 ; push dummy hi byte (must be 0 for char.)
         LDA  #$0D           ; ASCII CR is third argument
         PHA
         LDA  #0
         PHA                 ; another 0 for the hi byte
         LDY  #3             ; we're passing 3 arguments
         JSR  PUT            ; display all three arguments
         ...
ERRMSG   DB   'Illegal character: ',0  ;0-byte terminates the string
```

From inspection of the program fragment above, you may have surmised how the PROMAL LIBRARY routine PUT tells the difference between a single character argument and a string argument. If the argument is less than 256 (high byte is 0), then it is a single character. If the argument is greater than 256, then it must be the address of the string to print.

LOADING NON-RELOCATABLE MACHINE LANGUAGE PROGRAMS FROM WITHIN A PROGRAM

Function MLGET can be used to load a standard Apple or Commodore format machine language program, such as would be generated by commercial assemblers. You can specify whether you want the program loaded at the same location it was saved at, or at another location. Function MLGET is described in the LIBRARY MANUAL.

MAKING YOUR ASSEMBLY PROGRAMS RELOCATABLE

The RELOCATE program supplied on the PROMAL disk is capable of converting virtually any assembly language program into a relocatable PROMAL module. The advantages of doing this instead of simply using MLGET to load a standard, non-relocatable program are:

1. The program can be executed by simply typing its name from the PROMAL executive, just like any other PROMAL program.

2. The PROMAL loader will find a free location in memory to run the program automatically.

3. The program can be loaded under program control using the LOADer.

4. You can import variables and subroutines from your machine language package to PROMAL programs which call it.

This makes using RELOCATE the most desirable method of preparing large assembly language modules for use with PROMAL.

To use RELOCATE, follow these steps:

1. Prepare your assembly language source program in the usual way, following the interfacing guidelines in the preceding section, and the organizational guidelines suggested in the following section.

2. Assemble your program twice, once with the origin set at some arbitrary page boundary (greater than $0200), and once with the origin set exactly $0100 bytes higher in memory. Save both resulting object programs. You may use virtually any assembler you wish. A public domain PROMAL assembler is available from SMA. The program should consist of a single, contiguous block of code. Your zero-page variables will not be relocatable, and must be assigned locations as described in the foregoing section.

3. Execute RELOCATE from the PROMAL EXECUTIVE by typing:

```
RELOCATE Object Object0100 Objmodule
```

where Object and Object0100 are the names of the two machine language object files saved from the previous step, and Objmodule is the name of the desired PROMAL module to be generated as output. No default extensions are assumed for the first two file names, which are normally "PRG" type files for the Commodore and "BIN" type files for the Apple. The last filename will have a .C extension by default. If you want Objmodule to be an overlay instead of a program (see the section on the PROMAL loader for more information), you can specify an optional fourth argument as the single character O (the letter "oh").

4. When RELOCATE finishes, your program is ready to run or load.

ORGANIZING YOUR RELOCATABLE PROGRAM

Your assembly language package can have multiple procedures and functions in it which can be called from your PROMAL program, complete with passed arguments. In order for the LOADer to be able to link up your PROMAL program correctly with your finished relocatable machine language package, we suggest you follow some simple conventions, which we will illustrate in a skeletal example program.

To organize your program, decide which routines you will want to call from your PROMAL program. These should be entered by a jump table at the very start of your program. These JMPs should be followed immediately by any non-zero page variables which you wish to make available to the calling PROMAL program (often none will be needed). For example, if you want to be able to call three routines, and have one variable which can be accessed by other PROMAL programs:

```
TEMP    =      $00FE      ; Temp 0-page variable used by this program

        *=$1000           ; Dummy origin (make it $1100 for 2nd assembly)
FUNCA   JMP   FUNCA1      ; Function exported to PROMAL program
PROCB   JMP   PROCB1      ; Procedures exported to PROMAL program
PROCC   JMP   PROCC1

ANSPTR  .WORD      0      ; Variable exported to PROMAL program

FUNCA1  PLA
        STA   RA          ; Save return address
        ...               ; etc.
        RTS

PROCB1  PLA
        STA   RA
        ...
        RTS

PROCC1  PLA
        STA   RA
        ...
        RTS
        END
```

Assume that this program has assembled successfully. Now you want to export the definitions of your routines and your variable ANSPTR to the PROMAL program(s) which will be using your machine language package. Since the assembler can't generate an export file automatically, you can generate a "fake" export file by hand using the PROMAL EDITor. Assuming our sample package will be called MLPKG, you could generate this text file with the file name MLPKG.E:

```
IMPORT MLPKG ;10/16/85
  EXT ASM FUNC FUNCA AT $0000
  EXT ASM PROC PROCB AT $0003
  EXT ASM PROC PROCF AT $0006
  EXT WORD ANSPTR AT $0009
```

Be sure to start the IMPORT line exactly in column 1 and to observe the indentation for all other lines. The addresses shown after "AT" in each of the lines should be relative to the start of your machine language program (each JMP instruction is 3 bytes long). If you use the jump table, you won't have to change this export file even if you make changes in the body of your machine language program later.

You can now INCLUDE MLPKG.E in any PROMAL programs that will call the machine language package, and compile them. Your PROMAL program should also have a "bootstrap" program to load the machine language package, as discussed in the section on the PROMAL LOADer. For example:

```
PROGRAM BOOTPROG OWN
INCLUDE LIBRARY
INCLUDE PROSYS
...
LOAD "MLPKG", LDNOGO
LOAD "CALLSML"
...
END

PROGRAM CALLSML   ; main module, calls MLPKG
INCLUDE LIBRARY
INCLUDE MLPKG.E   ; Export file from M/L package
...
WORD MYVAR[100]
WORD I
...
ANSPTR=MYVAR    ; Using variable imported from MLPKG
IF FUNCA(23-MYVAR[I]) < 100   ; Calling M/L function
  PROCB MYVAR[I+1], MYVAR[I+2] ; & M/L procedure
...
END
```

These two programs can then be separately compiled. The final step is to make our machine language package relocatable:

```
RELOCATE OBJECT OBJECT100 MLPKG.C
```

assuming OBJECT is the output file from assembling the program at $1000, and OBJECT100 is the object file resulting from assembling it at $1100.

To execute the program, type:

```
BOOTPROG
```

which will load the relocatable machine language module MLPKG into memory at some available location, load the main PROMAL program CALLSML into memory above it, link the function and procedure calls and variable references to the machine language package, and execute the program.

TECHNICAL NOTES ON RELOCATE

The source code for RELOCATE is provided on a PROMAL diskette. It uses conditional compilation for the Commodore and Apple II versions. You may therefore modify it to meet your needs if you have an assembler which produces object output files which are not compatible with RELOCATE.

The Commodore version assumes the object code files to be used as input to RELOCATE will be standard Commodore object files of type PRG, such as are generated by the BASIC SAVE command. This format consists of a word giving the starting address, followed by a memory image of the object program.

The Apple II version assumes a standard PRODOS object file with a file type of BIN, such as is generated with a BSAVE command. The Apple version of RELOCATE is more complex because the information about the starting location of the memory image is contained in the directory instead of the file itself. See the PRODOS Reference Manual for details.

The format of the PROMAL relocatable object module which is generated as output from RELOCATE is as follows:

Position	Field Name	Description
0	FHEAD	Header ID byte, set to $CE
1	FTYPE	Module type, $01 for M/L prog, $05 for M/L overlay.
2-3	FHCDBA	Nominal code base address (ORG where assembled)
4-5	-	Not used, set to 0000.
6-7	FHCDSZ	Code size in bytes of memory image. Do not include this header or relocation table in count.
8-D	-	Not used, set to 0.
E-10	FHDATE	Date of assembly, 1 byte each for day, month, year (year-1900 really), in that order.
11	-	Reserved. Set to $04.
12-1D	FHCOMD	Program name followed by $00 terminator, as it would appear on a PROGRAM line of a PROMAL program.
1E-1F	-	Not used, set to 0000.
20-n		The actual object code memory image. The size of this field is given by FHCDSZ above.
n+1-n+2		Reloc. Table header, set to 'R' followed by 'A'.
n+3-n+4		Count of number of bytes which follow.
n+5-end		List of words of addresses relative to the start of the memory image above, where relocations must be made. For example, If the memory image was assembled to start at $1000 and starts with a JMP $112B instruction, then the first entry in the list would be $0002 (indicating the high byte of the address portion of the JMP instruction will need to be modified when the program is loaded).

The RELOCATE utility can accept a fourth argument of O (the letter O, not zero), indicating that the output object file is an overlay instead of a program (the SIZE command will display a type of "AOV" in this case, for Assembly Overlay).

INTERRUPT SERVICE ROUTINES

Due to limitations imposed by the architecture of the 6502 processor, it is not practical to write interrupt service routines in PROMAL. However, you may write and use machine language service routines.

For the **Commodore 64**, your program should prepare for using interrupts as follows:

1. Turn off interrupts.
2. Save the contents of the interrupt vector at location $0314-0315 in another variable.
3. Install the address of your service routine in $0314-0315.
4. Enable interrupts.
5. Your service routine will be entered from initial Kernal interrupt processing via the vector at $0314. Your service routine may not use any library routines, Kernal routines, or any other software. It must preserve all the registers and the stack. If the interrupt is caused by the 1/60th second timer, you must do a jump indirect through the saved vector you extracted in step 2. Otherwise, you must restore the registers already pushed by the Commodore Kernal and do an RTI.

Because of the heavy usage of interrupts made by the Kernal, we recommend you avoid interrupts on the Commodore 64 unless absolutely essential.

For the **Apple II**, you may freely use interrupts in the normal manner. However, you may not call any Library routines in the service routine, because they (and the underlying PRODOS system) are not re-entrant. You should observe all the restrictions detailed in the Apple Reference manuals.

Correct operation of PROMAL with interrupt routines is entirely the responsibility of the programmer.

This page is intentionally left blank.

APPENDIX J

RECURSION AND FORWARD REFERENCES

The PROMAL Language fully supports recursion. In fact, the PROMAL COMPILER (which is a 2800 line PROMAL program) makes extensive use of recursion. To make full use of recursion, it is sometimes necessary to call a Procedure or Function before it is defined. This is permitted in PROMAL, as follows:

Prior to the first invocation of the routine to be forward referenced, declare it as an external (but not ASM), for example:

```
EXT FUNC BYTE EXP     ; Allow forward reference to Expression Parser
EXT PROC STATEMENT    ; Ditto for Statement processing routine.
```

You may then have forward references to the routine, by calling it in the normal manner, for example:

```
TYPE = EXP
STATEMENT ASSIGN, BYTETYPE
```

At the desired location, complete the normal declaration of the Procedure or Function, for example:

```
FUNC BYTE EXP
...
END

PROC STATEMENT
ARG WORD ASGNLOC
ARG BYTE RESULTTYPE
...
END
```

Additional calls to these routines may follow their definition in the normal fashion, if desired. Note that declaring a forward reference in this manner defeats the compiler's argument count checking and also its checking for undefined subroutines, so be careful.

NOTE: If you have the optional Developer's disk, file XREF.S illustrates an excellent example of the use recursion for searching a tree.

This page is intentionally left blank

APPENDIX K

REAL FUNCTION SUPPORT

A PROMAL Diskette includes a file called **REALFUNCS.S** which contains the complete source code for all of the following arithmetic functions:

Name	Description	Example
ATAN	Arctangent (returns angle in radians)	Y = ATAN(X)
COS	Trigonometric cosine (angle in radians)	Y = COS(X)
EXP	Exponential (e to the X power)	Y = EXP(X)
LOG	Natural logarithm (base e)	Y = LOG(X)
LOG10	Common logarithm (base 10)	Y = LOG10(X)
POWER	Power (X to the Y power)	Z = POWER(X,Y)
SIN	Trigonometric sine (angle in radians)	Y = SIN(X)
SQRT	Square root	Y = SQRT(X)
TAN	Trigonometric tangent (ang. in radians)	Y = TAN(X)

These functions all expect arguments of type REAL and return results of type REAL. They are provided in PROMAL source form instead of as built-in functions (as in BASIC) because:

1. Many programs do not need any of these functions. If your program doesn't need them, you do not have to have them in memory, which makes about 1.5 K bytes of additional memory available for things you **do** need.

2. If you do need these functions, you can simply put the statement

```
INCLUDE REALFUNCS
```

in your program, and they will be included in your compiled program (assuming you have copied the REALFUNCS.S to your Working diskette used for compilation). No other declarations are needed to use the functions.

3. If you only need one or two of the functions, you can use the Editor to extract just the functions you need and insert them into your program. This saves memory and decreases compilation time compared with including the entire REALFUNCS.S file. Note, however, that some of the functions call other functions internally. For example, SIN calls COS and LOG calls LOG2, so be sure to copy all needed routines.

4. You can examine and study how the source code works. The algorithms used depend heavily on Hart, et al, <u>Computer Approximations</u>, published by John Wiley and Sons in 1968 and reprinted in 1978 with corrections. Comments in the source code identify which algorithm was selected.

BASIC users will find most of these functions familiar, except for POWER, which replaces the BASIC operator "^". The POWER function is defined only for positive values of the first argument. All the functions are believed to give better precision than Commodore or Applesoft BASIC, often as much as two additional significant digits. Through normal range arguments, the functions can be relied on for about 9.5 significant digits (slightly less for POWER). Even though these functions provide greater precision and are written entirely in PROMAL, they usually still execute faster than their BASIC counterparts, which were implemented in hand-coded assembly language.

NOTE: PROMAL version 2.0 and earlier had function ABS in REALFUNCS.S. Version 2.1 has ABS in the standard LIBRARY for improved convenience and performance.

Also included on one of the PROMAL diskettes is a file called **FLOOR.S**. This contains the PROMAL function FLOOR, which has the form:

```
Realvar = FLOOR (X)
```

where **X** is a REAL value. FLOOR returns a REAL result which is equal to the largest integer less than or equal to the REAL argument. For example:

```
INCLUDE FLOOR.S
REAL X
...
X = FLOOR (100000.89)  ; Returns 100000.0
X = FLOOR (-3.8)       ;Returns -4.0
```

APPENDIX L

COMPATIBILITY ISSUES

One of the goals of the designers of PROMAL was to achieve a high degree of compatibility for PROMAL source programs on different kinds of computers while at the same time allowing users the freedom to take advantage of the special features of each supported computer. Obviously this entails some compromises. To achieve 100 percent compatibility, you can only support the "lowest common denominator" between machines. Clearly this is not a satisfactory approach. Instead, a standard Library of functions was developed, which is kept as similar as possible on all machines, but with additional system-dependent functions also provided in additional libraries.

This section describes the major differences between the Apple II/Commodore 64 versions of PROMAL (hereafter referred to as jointly as "6502 PROMAL") and the IBM PC and compatibles version (hereafter referred to as "IBM PROMAL"). The information is oriented towards the software developer wishing to "port" an existing 6502 program to the IBM, but is also useful for going from the IBM to the 6502.

MAJOR DIFFERENCES BETWEEN 6502 AND IBM VERSIONS OF PROMAL

1. There is no EXECUTIVE in IBM PROMAL. 6502 PROMAL includes an EXECUTIVE program which is a command shell similar to DOS on the IBM PC. There is no EXECUTIVE in the IBM version because the DOS shell provides these functions. Users of 6502 PROMAL should have little difficulty adjusting to DOS, since the EXECUTIVE and DOS are fairly similar.

2. IBM PROMAL does not support multiple programs in memory at once, since DOS does not support it. This generally presents no problem.

3. IBM PROMAL file names are limited to 8 characters plus a three character extension, because this is the DOS standard. IBM PROMAL supports full DOS path names.

4. IBM text files (including PROMAL source files) have lines terminated by **CR, LF** pairs, whereas 6502 PROMAL uses only **CR** terminators, in keeping with the conventions of the respective computers. This may cause some initial problems when porting source files from one machine to the other. When moving source files from 6502 systems to the IBM, you will need to write a small "filter" program to insert a linefeed ($0A) after each CR ($0D). More significantly, if your 6502 program uses statements such as **PUT CR,...** to generate an end-line, you will need to edit your source file to change this to **PUT NL....** Using NL is preferred since it is portable between either machine; in the 6502 Library, NL is defined as a single character, CR. In the IBM Library, it is defined as the string CR,LF. When using a statement such as **OUTPUT "#C..."**, you do not have to change the **#C** since this is defined as the appropriate newline sequence on either machine.

5. In 6502 PROMAL, the file handle returned by OPEN always points to the name of the file. This is not true for IBM PROMAL, because standard DOS file handles are returned, which are small integers, not addresses. This is normally of no consequence. However, if your program depended on the file

handle pointing to the name you will need to change it.

6. IBM PROMAL does not support the W, L, S, or K devices. However, you can open a file named "W". If you manipulate WPTR, WEOF, etc. directly in your program, you will need to change this.

7. I-O redirection operates somewhat differently in IBM PROMAL. DOS provides the I-O redirection, not PROMAL. The REDIRECT procedure is not supported. I-O redirection, when enabled using the > operator on the DOS command line, affects all screen output, not just output to the STDOUT handle. Also, note that **GETLF (STDIN,...)** does not support the PROMAL line editing features from the keyboard, but only the DOS line editing keys.

8. The LOAD procedure is not supported in the present version 1.9 of IBM PROMAL. In most cases this should not pose a significant hardship since the IBM has a much larger memory space available for running your program, so programs needing overlays in the 6502 version will not need them in the IBM version. It is possible to have one PROMAL program chain to another program using the DOSCALL procedure.

9. Naturally, any machine language calls, memory mapped registers, etc. used in your programs will not be portable.

10. Applications using the T device may need to be altered for use on the IBM PC. The TMODE utility is not supported; the DOS MODE command replaces this program. IBM PROMAL supports interrupt driven serial I/O.

11. If your program uses special keys (such as function keys), you will need to adjust the key codes as specified in **Appendix B**. Function key string substitution is still supported in the IBM version, but not from the DOS shell.

12. If your program uses embedded control keys to select reverse video mode, you will have to change this since the IBM does not support a control sequence for reverse video (unless you use ANSI.SYS as described in the DOS documentation). Functions are provided for setting video attributes.

13. The DIR function displays file names in a different format on the IBM PC, consistent with the DOS **DIR** command (/W option).

14. The line editing keys for use with GETL, EDLINE, and INLINE are somewhat different for the IBM version, consistent with normal key conventions for the IBM.

15. CARG[0] is not defined in the IBM version.

16. OPEN for IBM PROMAL does not have a default file extension (it is .C on the 6502 version).

17. The RENAME function cannot have wildcards in the IBM version. Complete path names are supported, and you can rename into a different directory.

18. IBM PROMAL 2.1 reserves the following additional key words: LONG, STRUC, UNION, SIZEOF, SEARCHLIB.

APPENDIX M

RELATIVE FILE SUPPORT ROUTINES FOR COMMODORE 64

PROMAL on the Commodore 64 treats all files as Commodore sequential (SEQ) type files, including programs, text and data. For many database and business applications, another type of file structure may be more suitable for rapid access to data. The Commodore 1541 disk drive has an undocumented but fairly widely-known ability to create and access files by "relative records". Your local computer store can probably provide books with information on using relative records with BASIC, such as The Anatomy of the 1541 Disk Drive, by Abacus Software.

The PROMAL System Diskette contains a file called REL_FILE.S which provides a set of PROMAL routines for using relative files from your program. A totally complete discussion of relative files is beyond the scope of this manual, but here is a brief description.

A relative file is organized into a number of fixed-length records. The size of all records in the file is the same, and is established when the file is opened. The record size can be from 1 to 254 characters. Records of 20 to 100 characters or so are typically used. For a database application, each record might be subdivided into fixed-length fields; for example, a customer name field, address field, etc. Once you have opened the relative file on disk, you initialize the file. Initializing the file allocates space on the disk for the number of records you specify and sets each record to "empty".

Once you have opened and initialized the file, you may write and read records by specifying the relative record number desired. Typically this record number corresponds to a sequential customer number or some other "key" number by which the file is to be accessed. The first record on the file is number 1 (not 0), the last record has a relative record number equal to the highest record number specified at initialization.

The REL_FILE.S file has the source for routines to open, initialize, read, write and delete relative files. Due to internal format differences, you may not read or write relative files as ordinary sequential files, or by using the Executive or Editor (exception: you may DELETE or RENAME relative files). In particular, if you try to TYPE or COPY a relative file from the Executive, you will get a "FILE NOT FOUND" error because the type of the file is not sequential. Do not use DYNODISK with Relative files.

To use the relative file routines, put the following statement in your program before the first reference to the routines:

INCLUDE REL_FILES

The following subroutines are provided:

PROC REL_OPEN OPEN RELATIVE FILE

USAGE: **REL_OPEN** Filename, Recsize

Procedure REL_OPEN opens a relative record file. **Filename** is a string containing the desired file name. This may be any legal Commodore filename, but we suggest you use a legal PROMAL file name with a ".R" extension. **Recsize** is an argument of type BYTE specifying the size of each record, which may be 1 to 254. It is the programmer's responsibility to insure that the file is opened with the same record size every time.

In planning your record size, remember that the record size should be 1 greater than the actual maximum number of characters you plan to use in the record, to allow for the Carriage Return (CR) terminator which will be appended automatically to each record on disk. The 1541 drive only allows one relative file to be open at a time. REL_OPEN must be called prior to any other relative file routines.

EXAMPLE:

```
CON RECSIZE=81                    ; Up to 80 chars in a record
DATA BYTE FILE="INVENTORY.R"      ; Filename to be opened
...
REL_OPEN FILE,RECSIZE             ; Open relative file for I/O
...
```

PROC REL_INIT INITIALIZE RELATIVE FILE

USAGE: **REL_INIT** Numrecs

Procedure REL_INIT initializes a previously opened relative record file and specifies the maximum number of records to be allocated. Each record is initialized to "empty" (a null string). **Numrecs** (type WORD) is the desired maximum number of records. If this number is large, the initialization could take several minutes. It is only necessary to initialize a relative file when it is first created (after opening it) or when enlarging the maximum number of allowable records. It is not necessary (or desirable) to initialize it each time you open it. To enlarge the file for additional records, you can call **REL_INIT** again with **Numrecs** specifying the new maximum. Records previously written will not be affected.

EXAMPLE:

```
CON RECSIZE=81                    ; Up to 80 chars in a record
CON NUMRECS=200
DATA BYTE FILE="INVENTORY.R"
...
REL_OPEN FILE,RECSIZE             ; open relative file
REL_INIT NUMRECS                  ; initialize file with null records
...
```

PROC REL_WRITE WRITE RECORD TO RELATIVE FILE

USAGE: **REL_WRITE** Recnum, Record

Procedure REL_WRITE is used to write a particular record in an open and initialized relative file. **Recnum** is the desired relative record number (type WORD), and **Record** is a string containing the text of the desired record. The string does not have to include a carriage return; one will be appended when the record is written to disk. The record written must not be longer than the record size which was specified when the file was open.

If the record was previously written, the new record replaces it in its entirety, even if the new record is shorter than the record it replaces. **Recnum** must be between 1 and the value specified for **Numrecs** when REL_INIT was called, inclusive. The string written should not contain a byte of $FF (255). Naturally it cannot contain any $00 bytes either since this is the string terminator in PROMAL.

EXAMPLE:
```
    CON RECSIZE=50 ; Up to 49 chars in a record
    CON NUMRECS=300
    BYTE LINE[81]
    BYTE INDEX
    WORD RECNUM
    DATA BYTE FILE="MYDATA.R"
    ...
    REL_OPEN FILE,RECSIZE
    ...
    PUT NL,"WRITE WHAT RECORD NUMBER ? "
    GETL LINE
    INDEX=STRVAL(LINE,#RECNUM)
    PUT NL,"CONTENT OF RECORD ?"
    GETL LINE
    REL_WRITE RECNUM, LINE
    ...
```

The program fragment above prompts for entry of a record number and a line of text to be the desired record. It then writes the record specified.

PROC REL_READ READ RECORD FROM RELATIVE FILE

USAGE: **REL_READ** Recnum, #Buffer

Procedure REL_READ reads a specified record from an open relative record file and copies it to a specified buffer. **Recnum** is the desired record number (type WORD), between 1 and the value of **Numrecs** specified when the file was initialized. **#Buffer** is the address of the desired buffer to hold the record, which should be at least as large as the record size specified when the file was opened. The CR terminating the record on disk is not returned in the buffer; it is replaced with a $00 byte so the buffer can be treated as a standard PROMAL string. A record which has never been written will return a

null string without error.

EXAMPLE:

```
CON RECSIZE=50 ; Up to 49 chars in a record
CON NUMRECS=300
BYTE LINE[81]
BYTE INDEX
WORD RECNUM
DATA BYTE FILE="MYDATA.R"
...
REL_OPEN FILE,RECSIZE
...
PUT NL,"READ WHAT RECORD NUMBER ? "
GETL LINE
INDEX=STRVAL(LINE,#RECNUM)
REL_READ RECNUM, LINE
PUT LINE,NL
...
```

The program fragment above prompts for a record number and displays it on the screen, followed by a carriage return.

PROC REL_DELETE DELETE RELATIVE FILE

USAGE: **REL_DELETE** Filename

Procedure REL_DELETE is used to delete **an entire relative record file.** The file should be closed when REL_DELETE is called. All records will be discarded and the file space reclaimed for future use on the disk. **Filename** is a string containing the name of the file. The message

"01, FILES SCRATCHED, 01, 00"

will be displayed on the screen. This is not an error.

EXAMPLE:

```
...
REL_DELETE "MYDATA.R"
...
```

PROC REL_CLOSE CLOSE RELATIVE FILE

USAGE: **REL_CLOSE**

Procedure REL_CLOSE closes the previously-opened relative record file. No error occurs if the file is not open. This procedure should be called before exiting from any program which has opened a relative file, or when done with the file. Note that it is normal for the red light on the 1541 drive to be on the entire time a file is open. Because it is important to properly close the file, it is suggested that CTRL-STOP not be used to exit from a program which has opened a relative file.

EXAMPLE:

```
...
REL_OPEN "MYDATA",40
...
REL_CLOSE
...
```

Note that if you INCLUDE both RS_232 and REL_FILES in a single program, you will get some duplicate identifier errors when you compile, because both packages use and declare some of the same Kernal entry points. To correct this situation, simply copy whichever of these files is second in your program to another file, and edit it to delete the duplicate declarations.

The REL_FILE package requires version 1.1 or later of PROMAL.

RELDEMO PROGRAM

The PROMAL System disk contains a file called RELDEMO.S. This file is a simple demonstration program for relative files using REL_FILE.S support. It opens a relative file called "TEST_REL.R" for up to 20 records of 40 characters each, and prompts you to read or write selected records. The first time you run RELDEMO, you should select "initialize" from the menu before reading or writing records. A menu option is provided to delete the entire file if you no longer want it on your disk.

You can study the RELDEMO.S program for more information about using relative files. Since the REL_FILE.S support package is provided in PROMAL source form, you may also wish to study it to see how to use PROMAL to interface to the Commodore Kernal routines. Advanced users may even wish to use the same techniques to write their own direct-access disk routines. If you do decide to write your own disk-support routines in PROMAL, please note the following:

1. The PROMAL nucleus **always has channel 15 open to the disk command/error channel.** The routine REL_CHECK in REL_FILE.S provides a way to read the error channel.

2. PROMAL allocates C-64 channels in ascending order, with the secondary address the same as the channel number. You should pick "high" channels and secondary addresses (9 or 10 recommended) to keep out of PROMAL's way.

This page is intentionally left blank

APPENDIX N

OPERATING SYSTEM NOTES

COMMODORE 64

PROMAL Version 2.0 has full support for two disk drives. These can be either two 1541-type drives (one as device 8 and one as device 9), or dual diskette drives such as the SD-2 by Micro Systems Development (MSD), with drive numbers 0 and 1 on Commodore device 8. The default drive is drive 0. Files in drive 1 are designated by a "1:" prefix as part of the file name.

If a drive number is not specified as a prefix, the default drive is always drive 0 (device 8). A prefix of "1:" will access drive 1 (device 9). You always "boot up" PROMAL from drive 0. When compiling programs with INCLUDE files, the INCLUDE file name may have a drive prefix.

As shipped from SMA, PROMAL is set up to use one 1541 drive. If you wish to use a dual drive MSD system, you should disable DYNODISK permanently and set the device numbers for both drives to 8. This should be done from BASIC as follows:

```
LOAD "PROMAL",8
POKE 3553,8   :rem makes logical drive 1 device 8, not 9
POKE 3554,128 :rem defeats DYNODISK permanently
```

Insert a formatted disk in drive 0 and type:

```
SAVE "PROMAL",8,1
```

Then put the PROMAL diskette back in and type:

```
RUN
```

After the system is booted up, copy the rest of the files to your new disk. You can use a commercial copier to do this if you wish, or use the COPY command.

ELAPSED TIME FUNCTION

On the Commodore 64, you can read the "jiffy" clock using this function, TIME, which returns the clock reading in "jiffies" (1 jiffy = 1/60th of a second) as a REAL number:

```
EXT BYTE THI AT $A0
EXT BYTE TMED AT $A1
EXT BYTE TLO AT $A2
FUNC REAL TIME
BEGIN
RETURN TMED:+<<8 + TLO +65536.*THI
END
```


APPLE II

PROMAL uses the Apple ProDOS operating system. Users who are accustomed to operating under DOS 3.3 will find a utility program on the ProDOS Utilities Diskette (available at your Apple Dealer) which can convert your existing text and data files to ProDOS format so that you may use them with PROMAL.

Some non-PROMAL programs for the Apple produce text files with the high-order bit of each character set. These files may be converted to standard ASCII files (as expected by the PROMAL editor and other PROMAL programs) by using the CLEARBIT7 utility program on the PROMAL System diskette. The command syntax is:

CLEARBIT7 Oldfile Newfile

where **Oldfile** is the name of the file to be corrected and **Newfile** is the desired name for the corrected file to be written. The CLEARBIT7 utility also truncates any lines longer than 125 characters, so that the resulting **Newfile** will be acceptable to the PROMAL EDITOR.

SPECIAL PRODOS FUNCTIONS

The file PRODOSCALLS.S contains a source program fragment suitable for calling special ProDOS functions (such as testing or setting file attributes) which are not covered by built in LIBRARY routines. The ProDOS Technical Reference Manual contains all the neccessary details.

APPENDIX O

FORMATTING, BACKING UP, & MAKING WORKING DISKS

Apple II

One of the first things you should do with your all your PROMAL distribution disks is to **make at least one backup copy. Be sure to read the License agreement before opening your sealed diskette.** It is important to make a copy of your diskettes and only work with the copy, so that in the event that any files are accidentally deleted, you can always get a new copy from the original disk. Note that only the Demo disk is bootable. You may make backup copies for your own personal use subject to the license agreement. Making copies other than as permitted by the license agreement is a violation of Copyright Law and is a crime.

For the **Apple II**, you may back up your PROMAL diskettes using the ProDOS Utility Diskette which came with your Apple IIe or IIc (available from your Apple dealer). You should also use this program to format new blank diskettes before using them with PROMAL. It is a good idea to write the volume name of every diskette on the label using a soft marker. **All disks should have unique volume names.** To make a "working disk" which can boot up PROMAL and do development, use the ProDOS file copier or the PROMAL COPY or EXTCOPY commands to copy the following files from the PROMAL disk to your newly-formatted diskette:

```
PRODOS
PROMAL.SYSTEM
EDIT.C
EXECUTIVE.C
COMPILE.C       ; Either Demo compiler or full
LIBRARY.S
EXTDIR.C
COMPERRMSG.T
EXTCOPY.C       ; If desired
```

All the files above except the full compiler are on the Demo disk. COMPILE.C, EXTDIR.C, COMPERRMSG.T and EXTCOPY.C are not necessary to boot up the system, but you will usually want to have them on the disk if you will be developing any programs. Once your system is booted up, you can develop programs with a disk which only has COMPILE.C and COMPERRMSG.T (and perhaps EXTDIR.C) on it, to leave more room for your source and object programs. Remember to issue a PREFIX * command when changing disks. You may wish to copy other files such as PROSYS.S or REALFUNCS.S on an as-needed basis.

If you have a /RAM disk, you may want to set up a BOOTSCRIPT.J file to copy your working files to 0:. EXTCOPY is convenient for doing this.

It is a good practice to keep at least one formatted, blank diskette available at all times, with the Volume name clearly marked on the label. This will come in handy the first time you type in a big program with the Editor only to discover there's no room left on your working disk to save it! To extricate yourself, save to W and exit to the EXECUTIVE. Then use PREFIX * to select your blank disk and COPY W Filename to save your file on disk.

Commodore 64:

For the Commodore 64, the DISKETTE utility is a PROMAL program which provides the following disk and file maintenance services:

1. Duplicate an entire diskette using a single 1541/1571 disk drive, two 1541/1571 drives, or an MSD dual disk drive.
2. Format ("New") a diskette.
3. Copy a file to another diskette.
4. Erase files.
5. Rename a file.
6. Display file names (directory).
7. Change a diskette name or ID.

Several of these services such as copying, erasing, renaming and displaying the directory may also be done with built-in EXECUTIVE commands. The DISKETTE utility can copy, delete, and rename files of PRG type or SEQ type, with any legal Commodore name.

Probably your first use of the DISKETTE utility will be to make a backup copy of the PROMAL SYSTEM DISK or PROMAL DEVELOPER'S DISK. Be sure to read the License agreement before opening your sealed diskette(s). It is important to make a copy of all the PROMAL distribution diskettes and only work with the copy, so that in the event that any files are accidentally deleted, you can always get a new copy from the original disk. You may make backup copies for your own personal use subject to the license agreement. Making copies other than as permitted by the license agreement is a violation of US Copyright law and is a crime.

It is very easy to duplicate the PROMAL SYSTEM DISK, although rather time consuming because of the slow operation of the Commodore 1541 drive; it takes about 15-20 minutes to back up a PROMAL distribution disk. However, this is time well spent. If you are fortunate enough to have a dual drive system, it only takes 2 minutes. If you have a commercial fast-copier, you may use that. It is a good idea to put a write-protect tab on your PROMAL diskette before proceeding.

To run the DISKETTE utility, put the PROMAL Demo Disk (or a copy) in the drive and type this command from the EXECUTIVE:

```
--> DISKETTE
```

The screen will clear and a menu similar to this will be displayed:

```
PROMAL DISKETTE UTILITY 2.0

          MENU

Q = QUIT (TO EXECUTIVE)
D = DUPLICATE ENTIRE DISK
N = NEW (FORMAT) DISK
C = COPY A FILE
E = ERASE (DELETE) FILE(S)
F = FILE NAMES DISPLAY (DIRECTORY)
R = RENAME FILE
A = ALTER DISK NAME OR ID

YOUR SELECTION? _
```

Press D and RETURN to duplicate an entire disk. Then just follow the instructions. You will be asked if it is okay to unload the EDITOR to increase the size of the copy buffer; type Y and return. You may use the RETURN key by itself for a "yes" reply to questions needing a yes-or-no answer. You will be prompted when to swap disks if you have a single drive. When your duplicate disk is finished, the menu will be redisplayed. Press Q and RETURN to exit to the EXECUTIVE.

MAKING WORKING DISKETTES

For your normal operations, you will not need most of the files supplied on the PROMAL disks, but will want the PROMAL system so that you can "boot up" PROMAL from your working disk. The best way to do this is to use the DISKETTE utility to format a new disk and then copy only the files you want.

To format a disk using DISKETTE, select the N option from the menu and press RETURN. Again, just follow the directions. When prompted for a 2 character ID, pick any two characters that you have not used when formatting another disk. It is very important to use a different ID on each of your disks. This is how the 1541 disk drive DOS figures out when you have changed disks. If you have two diskettes with the same ID but different contents and swap them, the directories and files may be corrupted.

After formatting your new diskette, insert your copy of the PROMAL Demo disk in the drive and select the C option from the menu. Copy the following files one at a time using the C option:

```
PROMAL
EDIT.C
EXECUTIVE.C
COMPILE.C     ; Or the "full" compiler from the sealed disk
LIBRARY.S
COMPERRMSG.T
DATE.C
```

Note that unlike the EXECUTIVE COPY command, **you must type the ".C" extension explicitly** when copying files with DISKETTE. Also, since DISKETTE can copy files with any legal Commodore name (of type PRG or SEQ), **you must be careful to type in upper case letters if the file you want is in upper case** (you may wish to use CTRL-A to lock uppercase alphabetic characters).

The list of files above is the basic working set needed to boot PROMAL and develop software. You may also wish to copy DISKETTE.C, REALFUNCS.S or other programs. If you plan to boot up using another disk, then you only need to copy COMPILE.C and possibly EDIT.C if you will be compiling with the "B" (big program) option. It is possible to boot up without EDIT.C, COMPILE.C, COMPERRMSG.T, and DATE.C, but you will normally want these.

The size file which can be copied is limited to the size of the available buffer space, normally about 26K bytes. You can copy files of up to 64K bytes using the standard EXECUTIVE COPY command (more if you have 2 drives).

MISCELLANEOUS OPERATIONS

You can also delete and rename files with DISKETTE. When deleting, note that wildcards are acceptable when you are prompted for a file name to delete. Be very careful when using wildcards; there is no prompt for a chance to change your mind! When the deletion is completed, the standard Commodore disk message will indicate the number of files deleted ("scratched"). For example:

```
01, FILES SCRATCHED,02,00
```

indicates two files were deleted (the number after "FILES SCRATCHED", not before it!). If the message indicates 00 files scratched, you probably spelled the name wrong (don´t forget you have to add .C explicitly for PROMAL programs and match upper and lower case exactly).

APPENDIX P

PROMAL SYNTAX DIAGRAMS

The syntax diagrams on the following pages provide definitive reference for statement construction in PROMAL. If you are not familiar with syntax diagrams, then study the narrative which follows while referring to the named diagrams.

HOW TO READ SYNTAX DIAGRAMS

Consider the first path of the **STMT** diagram, which is the syntax diagram which shows you how to construct an assignment statement.

The symbols shown inside <u>ovals</u> are keywords or punctuation which must be typed exactly as shown. The symbols shown in <u>rectangles</u> describe things which you, the programmer, must supply. The lines connecting the ovals and rectangles show all the legal paths which you may take. For example, to make an assignment statement, the first thing you need is a VAR. This is the variable name to receive the result of the assignment. Exiting from the right side of the VAR. rectangle, we see that we have a "fork in the road", meaning we can take any of the paths. If we go "straight ahead" on the middle track, we come to an oval with an equals sign in it. Since this is an oval, we would write the equals sign. Finally, we come to a rectangular box called EXP. This stands for "expression", which means we can put any kind of expression there.

We already know about forming expressions. Probably the simplest expression is just a literal number. Therefore a legal assignment statement could be:

```
X=0
```

which sets the variable X to 0. We also know that a variable can be used for an expression, so another legal assignment statement would be:

```
ZVAL=X
```

We are assuming that X and ZVAL have been declared previously. We also know that more complicated expressions can be formed with operators. For example:

```
CMIN=(ZVAL-1)/2
```

But how do we find out EXACTLY what is a legal expression on the right side of an assignment statement? For example, is this legal?

```
VB=X OR Y
```

To find out, we consult the syntax diagram which defines an expression. Since you already know that expressions can be quite complicated and involve many possibilities, you might expect that the syntax diagram for an expression is also complicated. In fact, it is the most complex element of the language. Let's look at the EXP syntax diagram, which appears deceptively simple.

The EXP starts with something called a **RELATION.** Consulting the syntax diagram for a RELATION, we find it in turn starts with a SIMPLEXP. This is getting complicated! A SIMPLEXP can start either with a - sign or a TERM. Since our case doesn't start with a - sign, it must be a TERM (if it's legal). A TERM starts with something called a FACTOR.

A **FACTOR** starts with a lot of choices immediately. One of these choices (on the eighth line down) is "VAR.", which stands for variable. Now we are getting some place! We know X is a variable, but can our variable X be followed by the keyword "OR"? Follow the path out of the VAR rectangle, up and out of the diagram for FACTOR. We have now completed FACTOR, but we remember that FACTOR was just the start of a TERM, from the diagram above it. Tracing the path from the FACTOR rectangle we see several choices (*, /, etc.), but none of them are OR, so we continue to the right, exiting the TERM diagram.

Also, we remember that TERM was encountered in the SIMPLEXP diagram, so we follow the arrow out of the SIMPLEXP diagram, since there are no paths there that lead to OR. Returning to the diagram for RELATION after the SIMPLEXP box, we again find no path leading to an OR, so we exit to the right. Finally, we come back to the EXP diagram, and following the RELATION rectangle, we find a path that leads to OR. Therefore we know we can have a variable followed by OR. Following the path from the OR rectangle, we see we must come to RELATION again. Therefore we must have another RELATION after the OR. But since we already know from working our way down to FACTOR before that a relation can be a variable, we know that our statement is legal since Y is a variable.

Before finally concluding that our statement is legal, though, we must make sure that nothing else is required to follow what we already have. To do this, we must trace a path from the exit of the RELATION box to the exit of the EXP box, and then from the exit of the EXP box in the ASSIGN STMT diagram to the end-of-line symbol. The end-of-line symbol is shown as a down-pointing arrow in a circle, symbolically representing a carriage return.

Naturally you won't consult the syntax diagrams every time you write a statement! But if in the process of writing a program, if the compiler gives you an error message, and it is not obvious what is wrong, you can always consult the syntax diagrams to help find out what the problem is.

As an aside for the technically curious, you might be interested to know that the PROMAL COMPILER is really just a PROMAL program which tries to match your source program to the syntax diagrams! Each syntax diagram in this Appendix is implemented as one subroutine in the compiler. For instance, the COMPILER contains procedures called EXP, SIMPLEXP, RELATION, TERM, and FACTOR. Each of the "forks in the road" in the syntax diagram corresponds to an IF statement in one of these routines. To "parse" your assignment statement, the ASSIGNSTMT routine calls the EXP routine which calls the SIMPLEXP routine, etc., in the same manner as we just traced through the syntax diagram. If the compiler gets to a point where the next thing in your program doesn't match any of the choices, it prints an error message.

Now that you know how to read syntax diagrams, which are the "authority" on what is legal in PROMAL, you can refer to these diagrams whenever you have a question about the "legality" of a particular PROMAL statement.

DUMPFILE.S

This program is a useful utility which allows you to display the contents of any file in hex and ASCII, similar to the way the DUMP command displays memory. Any type of file can be dumped. The command syntax is:

```
DUMPFILE Filename [Type]
```

This will display the first 256 bytes of the file. The **Type** argument is needed only on the Commodore 64 for file types other than SEQ. No default file extension is used, so be sure to specify ".C" when dumping compiled programs. Pressing the RETURN key will display the next 256 bytes. Pressing any other key will terminate the program. DUMPFILE supports output redirection, so you can dump a file to the printer. The source and object code for DUMPFILE are on the Demo disk or the System disk. This program illustrates conditional compilation, so you must specify COMPILE DUMPFILE V=A on the Apple or V=C on the Commodore to compile the program. See the comments for more information.

FILECRC.S

This program computes the "Cyclic Redundancy Check" of a file and displays it. This is useful for comparing two files to see if they are identical. The comments in the source file explain the program operation and theory. It contains good examples of bit manipulation operators.

FIND.S

This program is discussed at length in the **MEET PROMAL!** manual. It searches a file for lines containing a specified string and displays these lines.

GRAPHDEMO.S **(COMMODORE 64 ONLY)**

This is a demonstration of high-resolution graphics using PROMAL. The program is self-explanatory when executed. The source code includes procedures for defining and clearing the hi-res screen, and drawing points and lines. These routines can be extracted using the editor for use with programs of your own design. NOTE: This program does **not** use or require the GRAPHICS TOOLBOX. The GRAPHICS TOOLBOX provides **much** higher performance and is much easier to use.

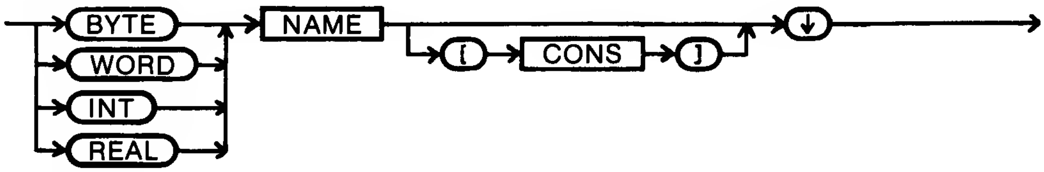
INFILTRATOR.S **(COMMODORE 64 ONLY)**

This is a fairly large and complex demonstration illustrating animation using screen scrolling, sprites, joystick input, and sound synthesis. You will need to issue a WS 0 command to edit this file and compile it using the B option (full compiler). This program is an excellent example of how to make good use of PROMAL procedures and functions to simplify a complex program.

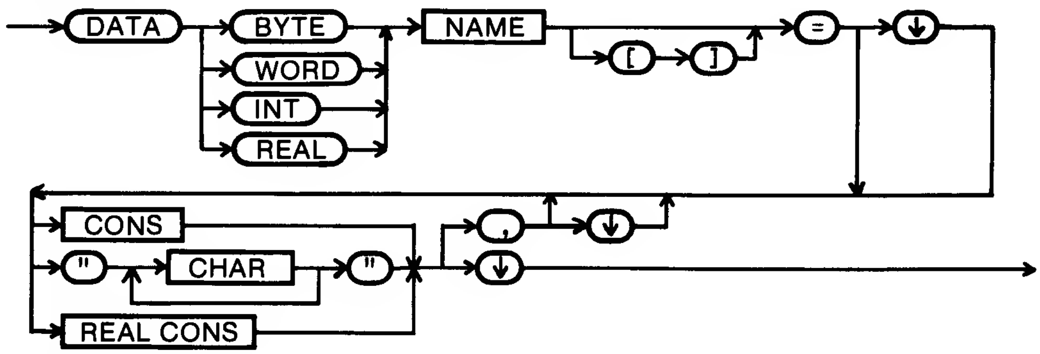
RELDEMO.S **(COMMODORE 64 ONLY)**

This program is a simple demonstration of relative files. It is explained in **Appendix M.**

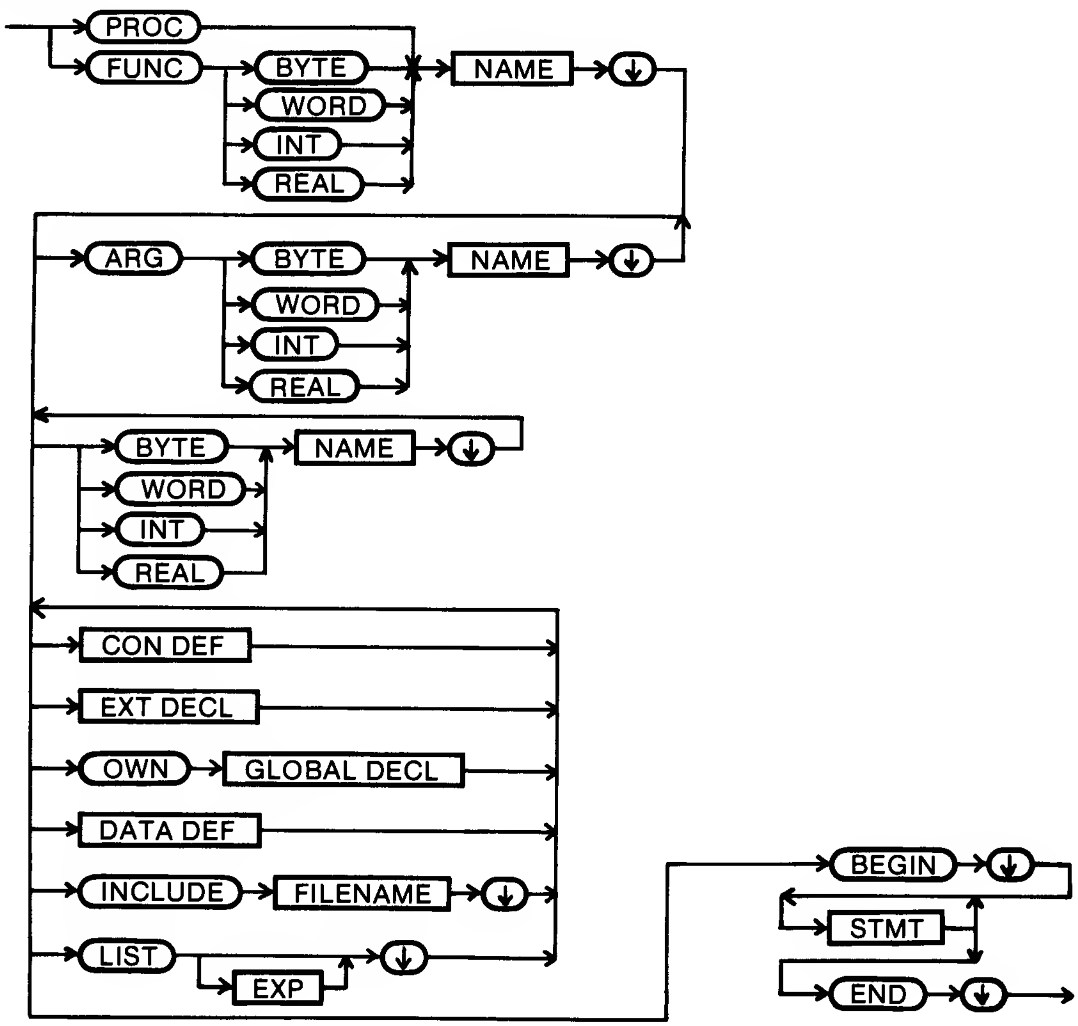
GLOBAL DECL.



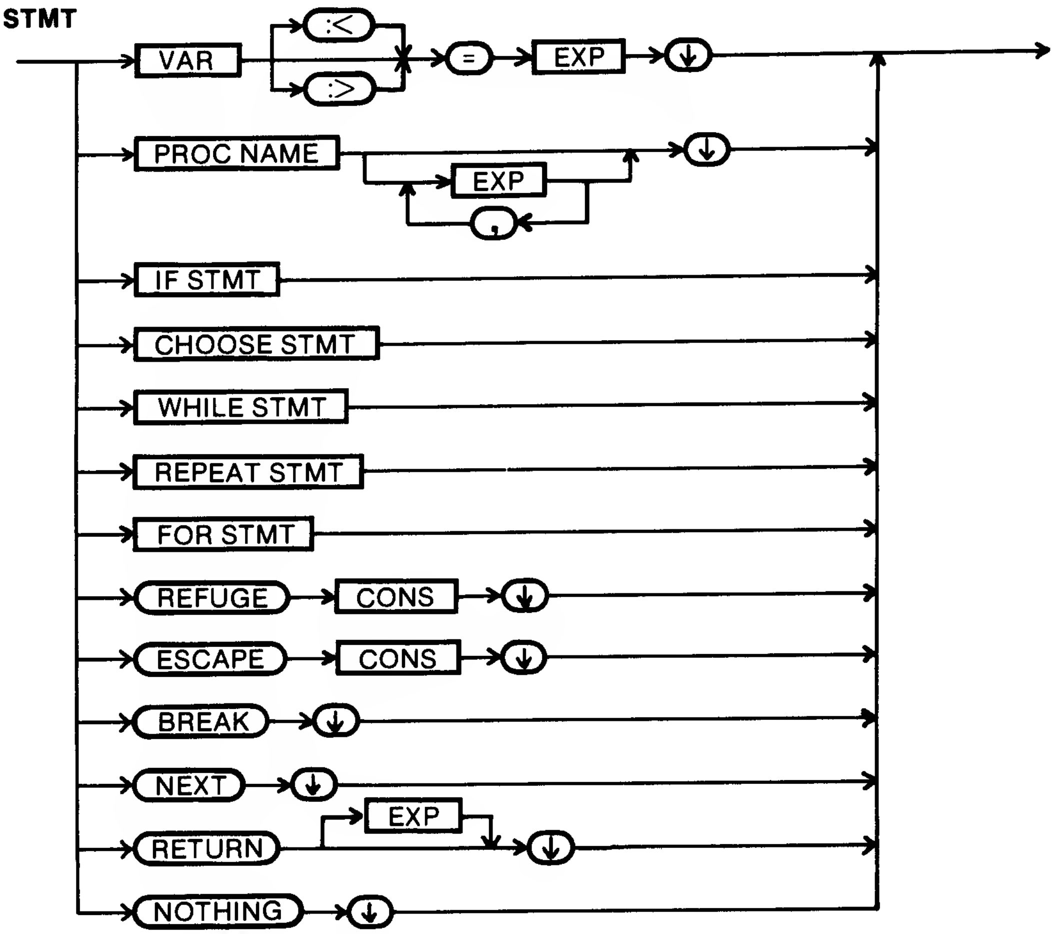
DATA DEF.



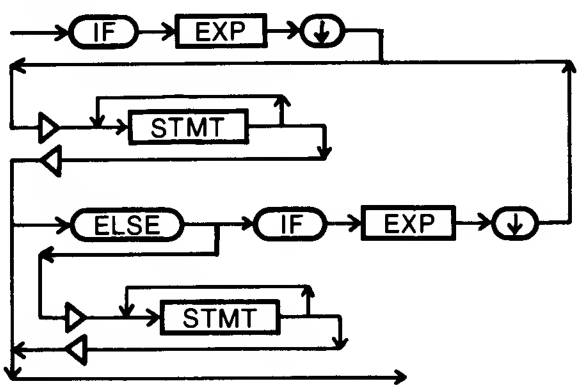
SUB. DEF.



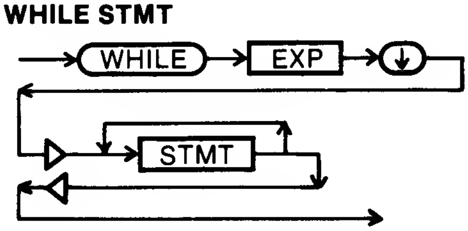
STMT



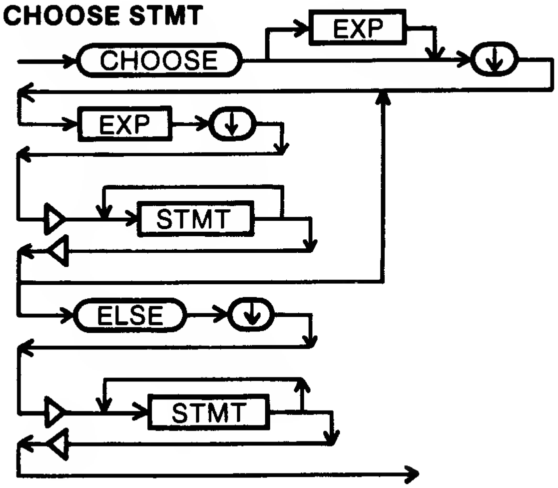
IF STMT



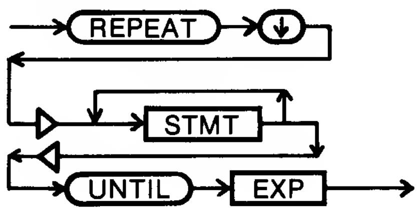
WHILE STMT



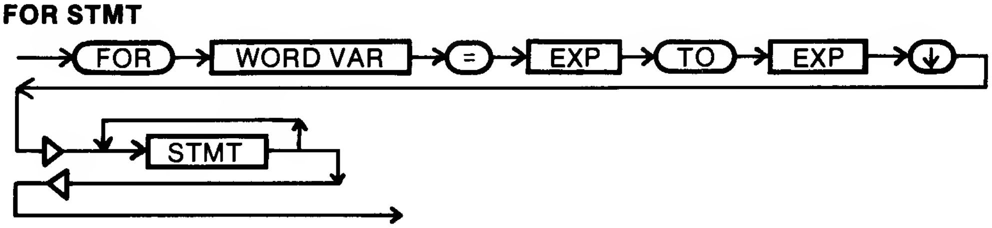
CHOOSE STMT



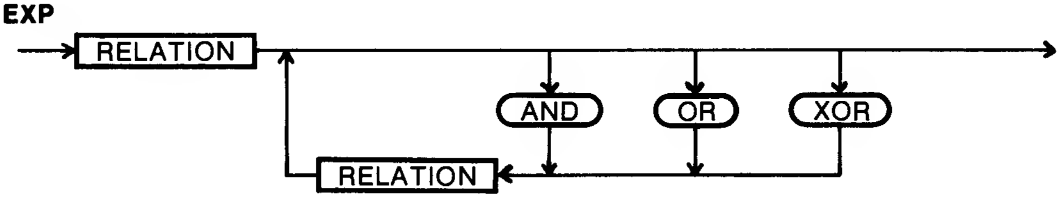
REPEAT STMT



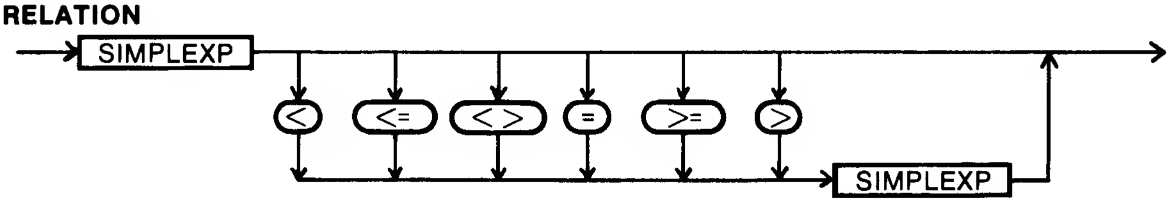
FOR STMT



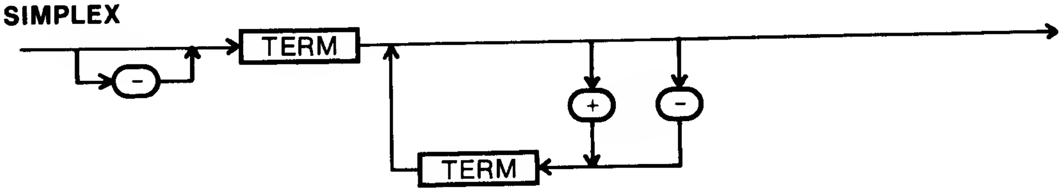
EXP



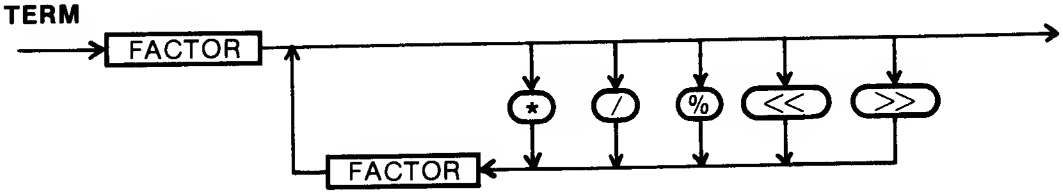
RELATION



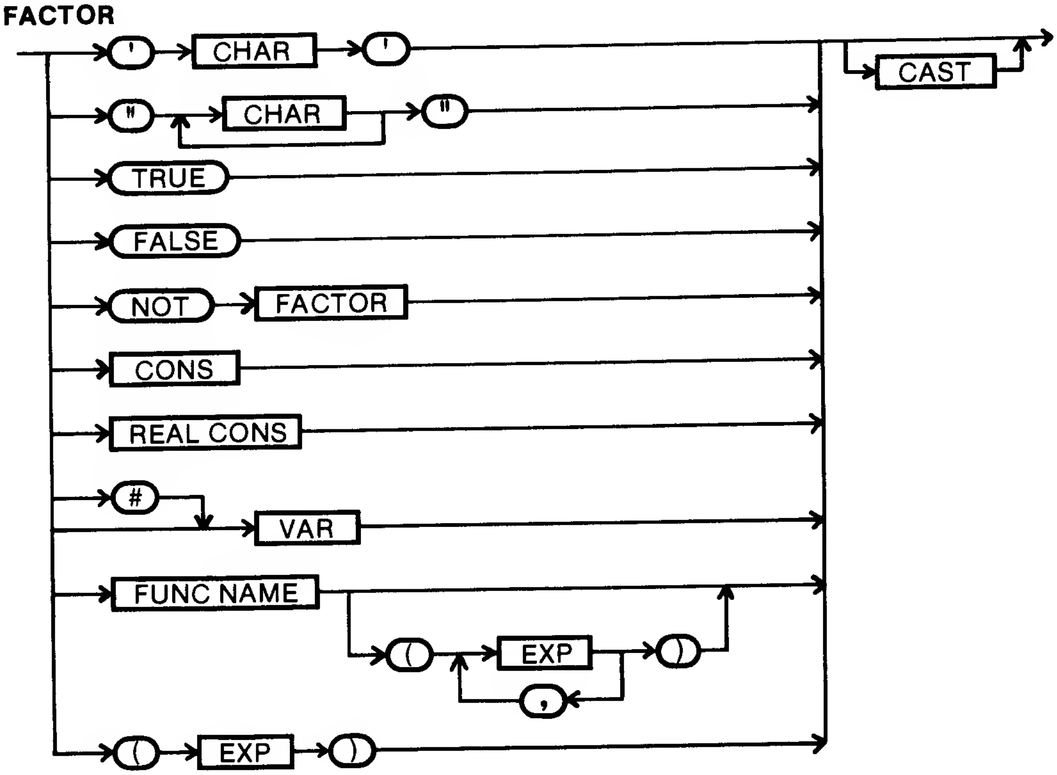
SIMPLEX



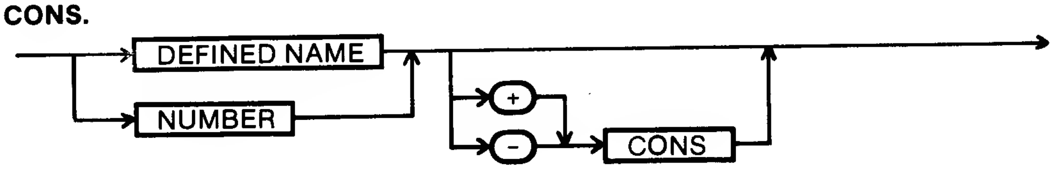
TERM



FACTOR

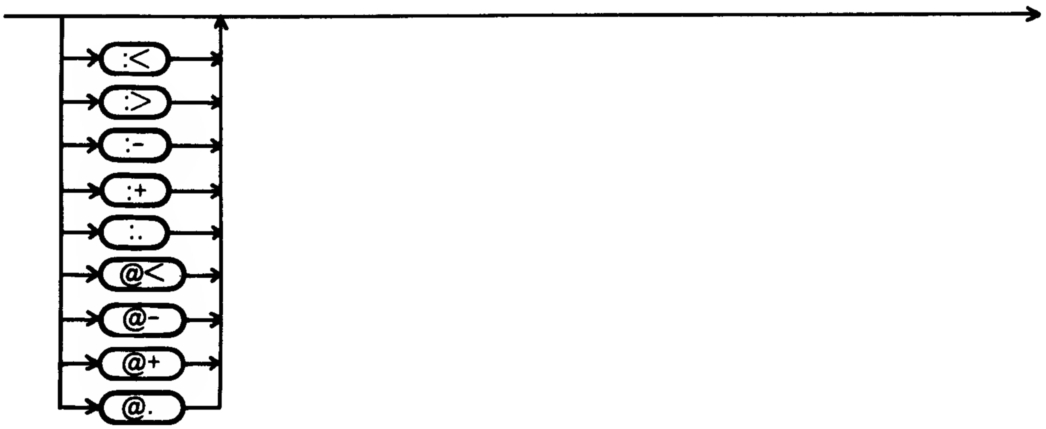


CONS.



VAR.

CAST



APPENDIX Q

PROMAL DEMO PROGRAMS

A number of demonstration programs are provided on the PROMAL System disk or optional Developer's disk, as well as on the Demo disk. Several demo programs were discussed in the **MEET PROMAL!** manual. You may compile and run these demonstrations, and you can use the EDITOR to extract parts of them to use in your own programs. By studying the programs you can learn many valuable techniques. Below is a summary of most of the demonstration programs provided.

BILLIARDS.S **(COMMODORE 64 ONLY)**

This program is discussed briefly in the **MEET PROMAL!** manual. It makes extensive use of sprites for animation of a billiards game. It also uses real math extensively for computing the motion of the balls, and has many conversions (type casts) between real and byte data types.

BUDGET.S

This is a very simple demonstration program which illustrates how to format real numbers for output. It uses the file BUDGETDATA.D for data. The file BUDGETDOC.T provides more information.

CALC.S

This program, when compiled, provides a demonstration program which simulates a four-function calculator with 26 memories (named A through Z). This program is discussed in **MEET PROMAL!**. To start the program, just type CALC, and follow the directions. The Demo disk has the source code for CALC, which illustrates how to write a recursive-descent expression evaluator in PROMAL. This program is only about 180 lines long (excluding comments), yet it can parse and evaluate arbitrary arithmetic expressions with nested parentheses.

CHECKSUM.S

This program computes the checksum of a specified block of memory. It is useful for determining if any bytes in a block of memory have changed. The comments provide further information on operation and theory.

CLEARBIT7.S **(APPLE II ONLY)**

This program will be found on the System Disk rather than on the demo disk. It is used to convert text files generated by other Apple II software which sets bit 7 of each byte to 1, to standard ASCII format for use with PROMAL. It also truncates lines longer than 125 columns, making files acceptable to the PROMAL EDITOR. It illustrates how to write a simple file filter in PROMAL.

DUMPFILE.S

This program is a useful utility which allows you to display the contents of any file in hex and ASCII, similar to the way the DUMP command displays memory. Any type of file can be dumped. The command syntax is:

```
DUMPFILE Filename [Type]
```

This will display the first 256 bytes of the file. The **Type** argument is needed only on the Commodore 64 for file types other than SEQ. No default file extension is used, so be sure to specify ".C" when dumping compiled programs. Pressing the RETURN key will display the next 256 bytes. Pressing any other key will terminate the program. DUMPFILE supports output redirection, so you can dump a file to the printer. The source and object code for DUMPFILE are on the Demo disk or the System disk. This program illustrates conditional compilation, so you must specify COMPILE DUMPFILE V=A on the Apple or V=C on the Commodore to compile the program. See the comments for more information.

FILECRC.S

This program computes the "Cyclic Redundancy Check" of a file and displays it. This is useful for comparing two files to see if they are identical. The comments in the source file explain the program operation and theory. It contains good examples of bit manipulation operators.

FIND.S

This program is discussed at length in the **MEET PROMAL!** manual. It searches a file for lines containing a specified string and displays these lines.

GRAPHDEMO.S **(COMMODORE 64 ONLY)**

This is a demonstration of high-resolution graphics using PROMAL. The program is self-explanatory when executed. The source code includes procedures for defining and clearing the hi-res screen, and drawing points and lines. These routines can be extracted using the editor for use with programs of your own design. NOTE: This program does **not** use or require the GRAPHICS TOOLBOX. The GRAPHICS TOOLBOX provides **much** higher performance and is much easier to use.

INFILTRATOR.S **(COMMODORE 64 ONLY)**

This is a fairly large and complex demonstration illustrating animation using screen scrolling, sprites, joystick input, and sound synthesis. You will need to issue a WS 0 command to edit this file and compile it using the B option (full compiler). This program is an excellent example of how to make good use of PROMAL procedures and functions to simplify a complex program.

RELDEMO.S **(COMMODORE 64 ONLY)**

This program is a simple demonstration of relative files. It is explained in **Appendix M.**

RELOCATE.S

This program converts machine language programs into relocatable form for use with the PROMAL loader. It is described in **Appendix I**. This program uses conditional compilation, so read the comments in the source before compiling. This program uses include files RELOCAPL.S or RELOCC64.S.

SORTDEMO.S **(APPLE II ONLY)**

This program is described briefly in **MEET PROMAL!**. It provides a demonstration of formatted output, file access, printer access, and general techniques.

SORTSTRING.S

This program provides a general shell sort routine for sorting arbitrary string arrays, and illustrates how to generate an array of strings read from a file. See the comments in the source file for usage.

SPLIT.S

This is a utility program which can be used to split a text file which is too large to edit into two smaller files. The file can be split after a specified number of lines, or before a line containing a specified string. The command syntax is:

```
    SPLIT Sourcefile Firstfile Secondfile Count
or
    SPLIT Sourcefile Firstfile Secondfile String
```

where **Sourcefile** is the file to be split, **Firstfile** is the name of the file to receive the first part of the file, and **Secondfile** is the name of the file to receive the other part of the file. No default file extensions are provided, so be sure to include ".S" for normal source files. **Count** is the number of lines to be copied from **Sourcefile** to **Firstfile**; the rest will go to **Secondfile**. **String** is a non-numeric string. The first line containing **String** anywhere in the line will be the first line sent to **Secondfile** when this form is specified. See the source program file for further information.

SSEND.S and SRECEIVE.S

This complementary pair of programs provides for error-free serial transmission of any type of files between Apple II computers at up to 9600 baud, and up to 600 baud for Commodore 64. Files can be exchanged between Apples and Commodores, too. The programs are described in **Appendix F**. The subroutines provided can be used as the basis for any kind of communications program.

TINYTERM.S

This program is a minimal implementation of a communications program for communicating over a modem to a time sharing service or bulletin board service. It is explained in **Appendix F**.

This page is intentionally left blank.

INDEX

A

ABORT (PROC).......................4-5
ABS (FUNC).........................4-5
ALPHA (FUNC).......................4-6
AND operator.............3-17,3-21/22
ARG statement......................3-42
Arguments:
-- cmmd line..1-14/15,1-21/22,3-56/57
-- defining....................3-42
-- passing..................3-42/45
-- substituting in JOB file......2-31
Arithmetic expressions.........3-17/20
Arithmetic operators..............3-17
Arrays:
-- of DATA strings.......1-27,3-15/16
-- of strings...................3-65
-- declaring.................3-14/15
-- multi-dimensional........3-15,3-64
ASCII character set................A-1
ASM routine, declaring..I-7/10,I-12/13
Assembly language subroutines:
-- calling LIB routines from.....I-10
-- interfacing to.............I-1/15
-- relocatable...............I-11/15
Assignment statement.........1-19,3-26
AT keyword in EXT statement..1-29,3-58
ATAN (FUNC)........................K-1
Audience, for PROMAL...............1-3

B

Backing up disks.................0-1/4
Batch job capability...........2-30/31
BEGIN statement...................3-41
BILLIARDS.S (sample program).......Q-1
Blank lines, as comments..........1-16
BLKMOV (PROC)......................4-6
BOOTSCRIPT.J file.................2-30
Bootstrap, to control loading.....3-76
BREAK statement................3-31/32
BUDGET.S (sample program).....1-24,Q-1
BUFFERS (APPLE II EXEC cmd).......2-15
Built-in functions & procedures...3-36
BYTE, data type............1-18,3-8/9

C

CARG variable..1-14/15,1-21/22,3-56/57
CALC.S (sample program)....1-24/25,Q-1
Characters, literal...............3-10
Characters, string extraction.....3-23
Checksum....................1-13,4-7
Checksum, MAP display.............1-13
CHECKSUM.S (sample program)........Q-1
CHKSUM (FUNC)......................4-7
CHOOSE statement..............3-30/31
CLEARBIT7.S (utility program)..N-2,Q-1
Clearing screen from program.4-43,4-45
CLOSE (PROC).......................4-7
CMPSTR (FUNC)......................4-8
COLOR (EXEC cmd)..................2-16
Command line args......1-14/15,3-56/57
Commands: (see EXECUTIVE, EDITOR)
-- user defined..................2-4
-- case insensitivity.............2-4
-- line editing, (TABLE 1)......2-5/7
-- notation conventions...........2-9
-- system-dependent keys..........2-7
-- EDITOR (TABLE 5)...........2-47/49
-- EXECUTIVE (TABLE 2)............2-8
Comment lines.....................1-16
Compatibility with IBM PROMAL....L-1/2
COMPILE (EXEC cmd).............2-56/58
Compiler:..................2-2,2-56/60
-- command options............2-56/58
-- cross reference utility.......2-60
-- dialog.....................2-58/60
-- edit after error..............2-59
-- introduction to...............2-56
-- invoking.................1-11,2-56
-- screen displays...............1-11
Compiling, conditional.........3-49/50
Compiling, sample program.........1-11
Compiling, very large prgrms...2-57/58
CON statement...............1-26,3-13
Conditional compilation........3-49/50
Conditional stmts, short-cuts.....3-33
Constants, defining.........1-26,3-13
COPY (EXEC cmd)................2-17/19
Copyright Notice...............ii,1-2
COS (FUNC).........................K-1
CS (EXEC cmd).....................2-19
CTRL ^ (Adj. rt.-APPLE II)...1-28,2-48
CTRL \ (Clr. end -APPLE II)...2-5,2-48
CTRL [(Start of line -C64)...2-6,2-48
CTRL A (Alphalock)........1-9,2-6,2-48
CTRL B (Cmd recall).......1-9,2-6,2-49
CTRL C (Abort cmd. -APPLE II)......2-7
CTRL D (Del char.-APPLE II)...2-5,2-47
CTRL E (Insert -APPLE II).....2-5,2-47
CTRL F (Strt of ln -APPLE II).2-6,2-48
CTRL I (Indent)...................2-48
CTRL J (Adj. rt. -C64).......1-28,2-48
CTRL K (Clr. end -C64)........2-5,2-48
CTRL L (End of ln -APPLE II)..2-5,2-48
CTRL N (Next page)................2-48
CTRL O (Adjust left)........1-28,2-49
CTRL P (Previous page)............2-48
CTRL Q (Un-indent -APPLE II)......2-48
CTRL RESET (Abort cmd.-APPLE II)...2-6
CTRL STOP (Abort cmd.-C64).........2-6

INDEX

CTRL U (Un-indent -C64)...........2-48
CTRL V (Normalize window -C64)....2-49
CTRL W (Set window -C64)..........2-49
CTRL X (Clr line).............2-5,2-48
CTRL Y (End of line -C64).....2-5,2-48
CTRL Y (Home -APPLE II)...........2-48
CTRL Z (End of file mark)..........2-6
CTRL <-- (Del char. -C64).....2-5,2-47
CURCOL (FUNC)......................4-9
CURLINE (FUNC).....................4-9
CURSET (PROC).....................4-10

D

DATA:
-- arrays of strings........1-27,3-16
-- defining arrays of.........3-15/16
-- definition...................3-15
-- REAL.........................3-15
-- statement example.....1-27,3-15/16
Data communications support......F-1/6
Data types......................3-8/9
Date, entering.....................1-6
DATE (EXEC cmd)...................2-20
DEL key (C64).................2-5,2-47
Delete key (APPLE II).........2-5,2-47
DELETE (EXEC cmd).............2-20/21
Demo diskette, limitations of......1-3
Demo programs....................Q-1/3
Device names (TABLE 4)............2-13
Device numbers (C64)...........E-1,N-1
DIR (FUNC)......................4-10/11
DIROPEN (FUNC)....................4-11
DISKCMD (EXEC cmd)............2-21/22
Disk drives, dual support of.......N-1
DISKETTE utility (C64)...........O-2/4
DUMP command, example..............1-8
DUMP (EXEC cmd)...................2-23
DUMPFILE.S (utility program).......Q-2
Dynamic memory allocation..........H-1
DYNO (EXEC cmd)...............2-22/23
DYNODISK.............1-5,2-22/23,4-18

E

EDIT (EXEC cmd)..............2-24,2-44
Editor:................1-9,2-2,2-44/55
-- CHANGE (F6).............1-26,2-51
-- COPY (F7)....................2-52
-- char. sets & modes (C-64).....2-55
-- cut & paste operations........2-52
-- DEL LN (F1)..................2-49
-- display format.............2-44/46
-- EDIT (EXEC cmd)..........2-24,2-44
-- edit buffer & workspace....2-54/55
-- editing keys (TABLE 5).....2-47/49
-- entering from EXECUTIVE.......2-44
-- features of..................2-44
-- FIND (F5)..........1-25/26,2-50/51
-- FKEYs legend after MARK..1-27,2-52
-- HELP (F7)....................2-46
-- initial screen display........2-45
-- inserting & deleting lines....2-49
-- inserting block or file....2-52/53
-- INS LN (F2)..................2-49
-- introduction to.............1-9/10
-- invoking............1-9,2-24,2-44
-- MARK (F3)................1-27,2-52
-- MOVE (F6)....................2-52
-- QUIT display............1-10,2-53
-- QUIT (F8)....................2-53
-- RECALL (F4)..............2-49,2-53
-- sample sessions.....1-9/10,1-25/29
-- saving block to file.......2-52/53
-- scrolling....................2-46
-- search & replace.........1-26,2-51
-- searching.................2-50/51
-- status area..................2-46
-- WRITE (F4)................2-52/53
EDLINE (FUNC).................4-12/14
Error messages..................C-1/12
Errors:
-- from LOADer...................3-71
-- after OPEN....................4-36
Executing sample program..........1-12
Executive:.....................2-2/43
-- arguments for commands.2-9,2-12/13
-- arg passing fm cmd line....3-56/57
-- commands, search order.........2-4
-- commands, summary (TABLE 2)....2-8
-- entering commands............1-6/7
-- guided tour of...............1-6/9
-- HELP screen...................1-8
-- line editing keys............2-5/6
-- user defined commands..........2-4
EXIT (PROC).......................4-14
EXP (FUNC)........................K-1
EXPORT Keyword....................3-73
EXPORT (.E) files..............3-74/75
Exporting, definition..........3-73/74
Expressions:
-- arithmetic................3-17/20
-- logical...................3-21/22
-- mixed mode................3-18/20
-- relational...................3-21
EXT keyword..............1-29,3-57/59
EXTCOPY (utility command).........2-18
EXTDIR: (utility command).....1-7,2-25
-- example of...................2-25
Extensions, file name.............2-12

INDEX

F

FALSE (0)....................1-21,3-21
FILECRC.S (sample program).........Q-2
Field spec., formatted output.....4-41
File descriptor..................3-51
File name extensions (TABLE 3)....2-12
File names, rules for:
-- Commodore 64.................2-10
-- Apple IIe/IIc.............2-11/12
Files:
-- converting APPLE II DOS 3.3....N-2
-- COPY (EXEC cmd)............2-17/19
-- DELETE (EXEC cmd)..........2-20/21
-- DISKETTE utility.............O-2/4
-- handle, definition of....1-22,3-51
-- JOB (.J)..................2-30/31
-- locked.......................2-32
-- opening...........3-51/52,4-36/40
-- RENAME (EXEC cmd).............2-39
-- TYPE (EXEC cmd)...............2-41
FILES (EXEC cmd)...............2-24/25
FILL (EXEC cmd)..................2-26
FILL (PROC)...................4-14/15
FIND.S (sample program)....1-15/23,Q-2
FKEY (EXEC cmd)...............2-26/27
FKEYGET (PROC)................4-15/16
FKEYSET (PROC)...................4-16
FOR statement.................3-29/30
Format string, output spec...3-37,4-41
Formatted output.......3-37/38,4-41/43
Formatting disks...............O-1/3
Forward references................J-1
FUNC, function header............3-41
Function keys:
-- default setting....1-6,2-3,2-26/27
-- editor's display..............1-9
-- redefining...........2-26/27,4-16
-- use of........................2-4
Functions & procedures:...1-18,3-36/47
-- arguments in..............3-42/45
-- built-in..................3-36/37
-- intro to.....................3-36
-- REAL........................K-1/2

G

GET (EXEC cmd)................2-27/29
GETARGS (FUNC)...................4-17
GETBLKF (FUNC)...................4-18
GETC (FUNC)......................4-19
GETCF (FUNC).....................4-20
GETKEY (FUNC)...........4-20/21,B-1/4
GETL (PROC)...................4-21/22
GETLF (FUNC)..................4-22/23
GETPOSF (FUNC) (APPLE II).........4-23
GETTST (FUNC)....................4-24
GETVER (FUNC).................4-24/25
GO (EXEC cmd).................2-28/29
GRAPHDEMO.S (sample program).......Q-2
Graphics, Hi-Res.........1-33,2-15,Q-2

H

Hardware requirements.............1-2
HELP display screen...........1-8,2-29
HELP (EXEC cmd)...............2-29/30
Hexadecimal, literal numbers....3-9/10
Hexadecimal, used in EXEC cmds....2-12
Hi-res graphics..........1-33,2-15,Q-2
HOME key (C64)...................2-48

I

IBM PROMAL compatibility.........L-1/2
IF statement..................3-26/28
IMPORT keyword...................3-74
Importing, definitions.........3-74/75
INCLUDE statement..1-11,3-37,3-48,3-74
Indentation............1-20,1-28,3-27
Indirect operators.............3-23/24
INFILTRATOR (sample program)..1-32,Q-2
Initialization, PROMAL system....2-3/4
INLINE (FUNC)....................4-25
INLIST (FUNC).................4-25/27
Input, numeric................3-39/41
Input, simple.................3-38/39
INSET (FUNC)..................4-27/28
INST key (C64)...............2-5,2-47
INT, data type...................3-8/9
Interfacing:..................3-51/60
-- to C64 graphics & sound....3-58/59
Interrupt service routines........I-15
INTSTR (PROC)....................4-28
IOERROR:
-- error code variable...3-51/52,4-36
-- errors, codes for.............4-36
I/O:
-- functions GETLF, PUTF......3-52/53
-- redirection..................2-14
-- redirection, example of..1-23,2-14
-- with files STDIN/STDOUT....3-53/54

J

JOB (EXEC cmd)............2-30/31,2-33
JOB files (.J):................2-30/31
-- substitution arguments in.....2-31
JSR (PROC)..................4-29,I-1/4

K

Keyboard (K) device...............2-13
Key codes.......................B-1/4

L

LENSTR (FUNC)..................4-29/30
LIBRARY:............1-11,2-3,3-51,4-2
-- (L) device...................2-13
-- routine description notation...4-4
-- summary of routines..........4-2/3
Line editing, keys (TABLE 1).....2-5/6
LIST statement...................3-48
Literals:......................3-9/12
-- characters...................3-10
-- numbers....................3-9/10
-- strings....................3-10/11
LOAD (PROC)..............3-70/73,4-30
LOADER:......................3-67/82
-- bootstrap program with.....3-76/77
-- calling..................3-70/73
-- definitions used...........3-67/68
-- errors from..................3-71
-- EXPORTing definitions......3-73/74
-- IMPORTing definitions......3-74/75
-- memory diagram...........3-70,3-78
-- operation of..............3-68/70
-- options for..................3-72
-- overlays..................3-77/79
-- separate compilation.......3-75/76
Loading, PROMAL diskette.........1-4/6
Loading, programs......2-27/28,4-32/33
Local variables..........1-19,3-45/46
LOCK (EXEC cmd)..................2-32
Locked file.................2-32,2-42
LOG (FUNC).......................K-1
LOG10 (FUNC).....................K-1
Logical operators.............3-21/22
LOOKSTR (FUNC)...................4-30

M

M array.....................3-24,3-44
Machine language programs:
-- calling LIB routines from.....I-10
-- calling with JSR........4-29,I-1/4
-- embedded in DATA............I-4/6
-- effect of BRK..................I-3
-- executing with GO..........2-28/29
-- interfacing to.............I-1/15
-- interrupt service with........I-15
-- loading with GET...........2-27/28
-- loading with MLGET.........4-32/33
-- passing arguments to........I-8/10
-- relocatable...............I-11/15
MACRO (EXEC cmd)..............2-32/33
MAP (EXEC cmd):...........1-12,2-33/35
-- display screens...1-12/13,2-33/35
Memory allocation:
-- MAP display definitions....2-33/36
-- dynamic.......................H-1
Memory map, PROMAL internal......G-1/6
MAX (FUNC).......................4-31
MIN (FUNC).......................4-32
MLGET (FUNC)...................4-32/33
MOVSTR (PROC)..................4-33/34
MSD dual disk support (C64)........N-1

N

Names, rules for.................3-7/8
NCARG variable.........1-21/22,3-56/57
NEWDIR (EXEC cmd).................2-36
NEXT statement...................3-32
NOREAL (EXEC cmd)..............2-36/37
NOT operator.............3-17,3-21/22
Notation, conventions.............2-9
NOTHING statement.................3-32
Null (N) device................2-13/14
Numbers, literal................3-9/10
NUMERIC (FUNC)....................4-34
Numeric input..................3-39/41

O

O, compiler option............2-56/57
Object program.....................1-4
ONLINE (FUNC) (APPLE II)..........4-35
OPEN (FUNC).......1-22,3-51/55,4-36/40
OPEN, error codes.................4-36
Opening files..........3-51/52,4-36/40
Operating system notes...........N-1/2
Operators:
-- arithmetic.................3-17/18
-- indirect & address.........3-22/23
-- list of all..................3-17
-- logical...................3-21/22
-- relational....................3-21
-- shift........................3-22
Options for loader...............3-72
OR operator..............3-17,3-21/22
Output, field descriptors.........4-41
OUTPUT (PROC)..................4-41/43
OUTPUTF (PROC).................4-43/44
Output:
-- formatted numeric..3-37/38,4-41/44
-- PROC, example..............3-37/38
-- simple........................3-37
OVERLAY statement.........3-25,3-77/79
Overlays:
-- definition of.................3-68

-- using......................3-77/79
-- guidelines for............3-81/82
-- memory map...................3-78
-- sample program...............3-78
OWN variables...............3-46,H-1

P

Pathnames.........................2-11
PAUSE (EXEC cmd).................2-37
Pointers......................3-22/24
POWER (FUNC).......................K-1
PREFIX (EXEC cmd)...........2-11,2-38
Printer (P) device........2-13,3-54/55
Printer support..................E-1/2
PROC, procedure header............3-41
Procedures & functions:...1-18,3-36/47
-- introduction to...............3-36
-- passing arguments to.......3-42/45
-- local variables in.........3-45/46
ProDOS, special functions..........N-2
Program authors.....................ii
PROGRAM statement............3-4,3-25
Programs, demo:..................Q-1/3
-- BUDGET.......................1-24
-- CALC.........................1-25
-- FIND......................1-16/23
PROMAL:
-- definition of.................1-3
-- initialization..............2-3/4
-- loading.....................1-4/6
-- signon screen.................1-6
-- special capabilities..........1-29
-- system components.............2-2
-- vs. BASIC.....................1-4
PROMAL language:
-- applications of...............3-2
-- data types..................3-8/9
-- introduction to...............3-2
-- overview....................3-3/6
-- names.......................3-7/8
-- reserved words............3-7,L-2
-- rules for....................1-18
-- syntax diagrams.............P-3/8
PROQUIT (PROC)...................4-44
PUT (PROC)...........1-10,3-37,4-44/45
PUTBLKF (PROC)................4-45/46
PUTF (PROC).................1-22,4-47

Q

QUIT (EXEC cmd)...............2-38/39

R

RANDOM (FUNC)..................4-47/48
REAL:
-- constants disallowed..........3-13
-- DATA..........................3-15
-- data type....................3-8/9
-- literals......................3-10
-- numbers.......................1-24
-- variables, internal format.....D-2
REALSTR (PROC).................4-48/49
Recursion & forward references.....J-1
REDIRECT (PROC)................4-49/50
Redirection, I/O............1-23,2-14
REFUGE statement..............3-33/34
Relative file support (C64)......M-1/5
RELDEMO.S (sample program).....M-5,Q-2
RELOCATE utility..............I-11/15
RELOCATE.S (utility program).......Q-3
RENAME (EXEC cmd).................2-39
RENAME (FUNC).................4-50/51
REPEAT statement..................3-29
Reserved words................3-7,L-2
RETURN key...................2-5,2-46
RETURN statement..............3-41/42
Reverse video...........3-11,4-43,4-45
RS-232 support...................F-1/6
Runtime errors...................C-2/3
Runtime errors, locating...........D-1

S

Screen (S) device.................2-13
Scrolling, left & right......1-20,2-47
Scrolling, up & down.........1-16,2-47
SET (EXEC cmd)................2-39/40
SETPOSF (PROC) (APPLE II).........4-51
SETPREFIX (FUNC) (APPLE II).......4-52
Shift operators...................3-22
Signon display.....................1-6
SIN (FUNC).........................K-1
SIZE (EXEC cmd)...............2-40/41
SORTDEMO.S (sample prog.)..1-30/31,Q-3
SORTSTRING.S (sample program)......Q-3
SPLIT.S (utility program)..........Q-3
SRECEIVE.S (utility prog.)...F-4/5,Q-3
SSEND.S (utility program)....F-4/5,Q-3
SQRT (FUNC)........................K-1
Stack overflow....................3-47
Starting, system..........1-4/6,2-3/4
Statements:...................3-25/34
-- (=) assignment................3-26
-- BREAK......................3-31/32
-- CHOOSE.....................3-30/31
-- ESCAPE & REFUGE............3-33/34
-- FOR........................3-29/30

-- IF................1-21/22,3-26/28
-- NEXT.........................3-32
-- NOTHING......................3-32
-- PROGRAM......................3-25
-- REPEAT.......................3-29
-- RETURN....................3-41/42
-- WHILE................1-20,3-28/29
STDIN, SDTOUT file handles.....3-53/54
STOP key (C64)..................2-6/7
String operations:........3-45/46,3-23
-- arrays of....................3-65
-- compare...............1-20/21,4-8
-- conversion.......4-28,4-52/53,4-57
-- editing.............4-12/14,4-25
-- length...................4-29/30
-- move.....................4-33/34
-- searching........4-25/28,4-30,4-55
STRREAL (FUNC)................4-52/53
STRVAL (FUNC).................4-53/54
Subroutines:......1-18,3-41/42,3-46/47
-- passed arguments..........3-42/45
-- user defined..............3-41/47
Subscripts, arrays........3-14/15,3-64
SUBSTR (FUNC)...................4-55
Syntax diagrams, how to read.....P-1/2
Syntax diagrams.................P-3/8
System data areas...............G-5/6

T

TAN (FUNC)......................K-1
Telephone (T) device........2-13,F-1/6
TINYTERM.S (sample program)....F-5,Q-3
TO keyword......................3-29
TESTKEY (FUNC)...........4-55/56,B-1/4
TOUPPER (FUNC)..................4-56
Trademarks....................ii,1-2
TRUE (1)...................1-21,3-21
TYPE (EXEC cmd).................2-41

U

UNLOAD (EXEC cmd)...............2-42
UNLOCK (EXEC cmd)...............2-42
UNTIL keyword...................3-29
Unprintable codes, embedding......3-11
Upper & lower case mode (C64).....2-55
Upper case & graphics mode (C64)..2-55
User-defined commands.............2-4
User-defined subroutines.......3-41/47

V

Variables:
-- arrays, declaring.........3-14/15
-- command line argument......3-56/57
-- declarations..............3-12/13
-- external (EXT)........1-29,3-57/59
-- global..................1-19,3-44
-- initializing all to zero......3-66
-- introduction to rules.........1-18
-- local................1-19,3-45/46
-- locating in memory...........D-1/2
-- non-initialization of.........3-13
-- OWN..........................3-46
-- types supported.........1-18,3-8/9
Volume names (APPLE II)...........2-11

W

WHILE statement:...............3-28/29
-- example of...................1-20
WORD data type.............1-18,3-8/9
WORDSTR (PROC)...................4-57
Workspace:
-- and edit buffer............2-54/55
-- auto update after edit........2-54
-- changing size of.............2-43
-- clearing of..................2-43
-- writing to...................1-10
-- (W) device...................2-13
WS (EXEC cmd)....................2-43

X

XOR operator.............3-17,3-21/22
XREF (utility program)............2-60

Z

ZAPFILE (FUNC)................4-57/58

P R O M A L

(PROGRAMMERS MICRO APPLICATION LANGUAGE)

DEVELOPER'S GUIDE

For Apple II and Commodore 64 Computers

SYSTEMS MANAGEMENT ASSOCIATES, INC.
3325 Executive Drive
Raleigh, NC 27609
(919) 878-3600

Rev. C - Sep. 1986

TABLE OF CONTENTS

INTRODUCTION..3

GENERAL DESCRIPTION...3

SPECIAL CONSIDERATIONS..4

MEMORY MAP DIFFERENCES..5

WORKSPACE (W DEVICE) CONSIDERATIONS.............................7

SIZING YOUR APPLICATION PROGRAM.................................8

EXIT, ABORT, AND RUNTIME ERROR PROCESSING.......................9

SPECIFYING YOUR OWN ERROR RECOVERY..............................11

LOADING MACHINE LANGUAGE ROUTINES...............................11

BUILDING YOUR MASTER DISKETTE...................................12

COPY PROTECTION...12

MISCELLANEOUS...12

FINAL NOTES...13

TABLES & FIGURES

FIGURE 1: Memory Maps...5

TABLE 1: Runtime Error Codes....................................10

PROMAL DEVELOPER'S GUIDE

INTRODUCTION

With the PROMAL DEVELOPER'S SYSTEM and this manual, you will be able to generate diskettes containing your compiled PROMAL programs, which can be run on computers which **do not** have the PROMAL system. This will allow you to sell or distribute your application programs written in PROMAL to users who do not own PROMAL. As long as the acknowledgement requirements of the License Agreement are met, no royalties or other payments will be due to SMA, regardless of how many copies you sell or distribute.

If you have been using the PROMAL system, you already know that it is far superior to BASIC for developing programs, and that programs execute much faster. As a program author, you will also appreciate a very significant advantage of PROMAL: since you only need to distribute the compiled object code, you have a greater degree of security from unauthorized copies or modifications. You do not have to provide the source code.

By using the DEVELOPER'S SYSTEM, you will be able to make disks which "boot up" similar to the PROMAL Demo diskette, except that instead of running the PROMAL EXECUTIVE and EDITOR, your application program will be executed. You have control over what the programs are named and what program will be run. Your application can also load any other PROMAL or machine-language programs it might need.

GENERAL DESCRIPTION

The rest of this manual assumes that you already have a basic working familiarity with the PROMAL system.

When you "boot up" the regular PROMAL system, you LOAD a file called "PROMAL" and RUN it on the Commodore 64, or autoboot PROMAL.SYSTEM on the Apple II. This file contains the PROMAL nucleus and all the routines in the library. This file is called the runtime package. When this PROMAL runtime package is run, it automatically loads the EDITOR and EXECUTIVE, as well as the LIBRARY definitions, and begins execution of the EXECUTIVE.

The PROMAL DEVELOPER'S SYSTEM contains a special version of the runtime package which loads and runs your application program instead of the EXECUTIVE and EDITOR. You copy this runtime system and your application program onto a **MASTER DISKETTE** using a special program called **GENMASTER**. You may then duplicate and distribute your master diskette.

For the Apple II, your application will autoload on power-up just like PROMAL. For the Commodore 64, a user will LOAD the runtime package (which you can name anything you want) from the master diskette and RUN it. The runtime package will autoload your application program and execute it. No PROMAL signon message or other indication that this is a PROMAL program will be displayed; your program will be completely in control. Since there is no EXECUTIVE or EDITOR, when your program terminates (if it does), it will exit as described for procedure PROQUIT in the PROMAL Language Manual.

There is presently no provision for generating Commodore 64 cartridges or ROM-resident code.

The general procedure you will use for preparing an application program for distribution is:

1. Write, compile and test your application program using the regular PROMAL system.

2. When you are satisfied that your program is correct, use the GENMASTER program provided with the Developer's disk to bind your compiled application program with the special runtime package, and copy it to your desired master diskette.

3. Duplicate the master diskette and distribute it as you see fit. **You must include the acknowledgement notice: "(program) is a licensed PROMAL application program. PROMAL is a trademark of Systems Management Associates, Inc., Raleigh, NC.".** This notice must appear on the diskette label, in the documentation, or on an initial screen display of your program.

This manual only addresses Step 2. But before discussing the mechanics of Step 2, we will examine some of the special considerations which you should be aware of when preparing an application program for mastering.

SPECIAL CONSIDERATIONS

The main difference between running your application under the regular PROMAL system and as a dedicated application is that you will no longer have the EXECUTIVE available. Therefore:

1. The user will not be able to issue EXECUTIVE commands.

2. Your program cannot receive command arguments from the EXECUTIVE.

3. Your program cannot EXIT or ABORT to the EXECUTIVE (if you do EXIT or ABORT, the PROQUIT procedure will be executed).

4. If your program encounters a fatal programming error which would normally abort back to the EXECUTIVE (such as division by zero or calling a library routine with an illegal number of arguments), it will not return to the EXECUTIVE. A method is provided for you to recover from these errors within your application program (discussed below).

5. Since you have no EXECUTIVE to execute selected programs in memory, only your one application will be in memory at one time (unless you LOAD additional programs from within your program).

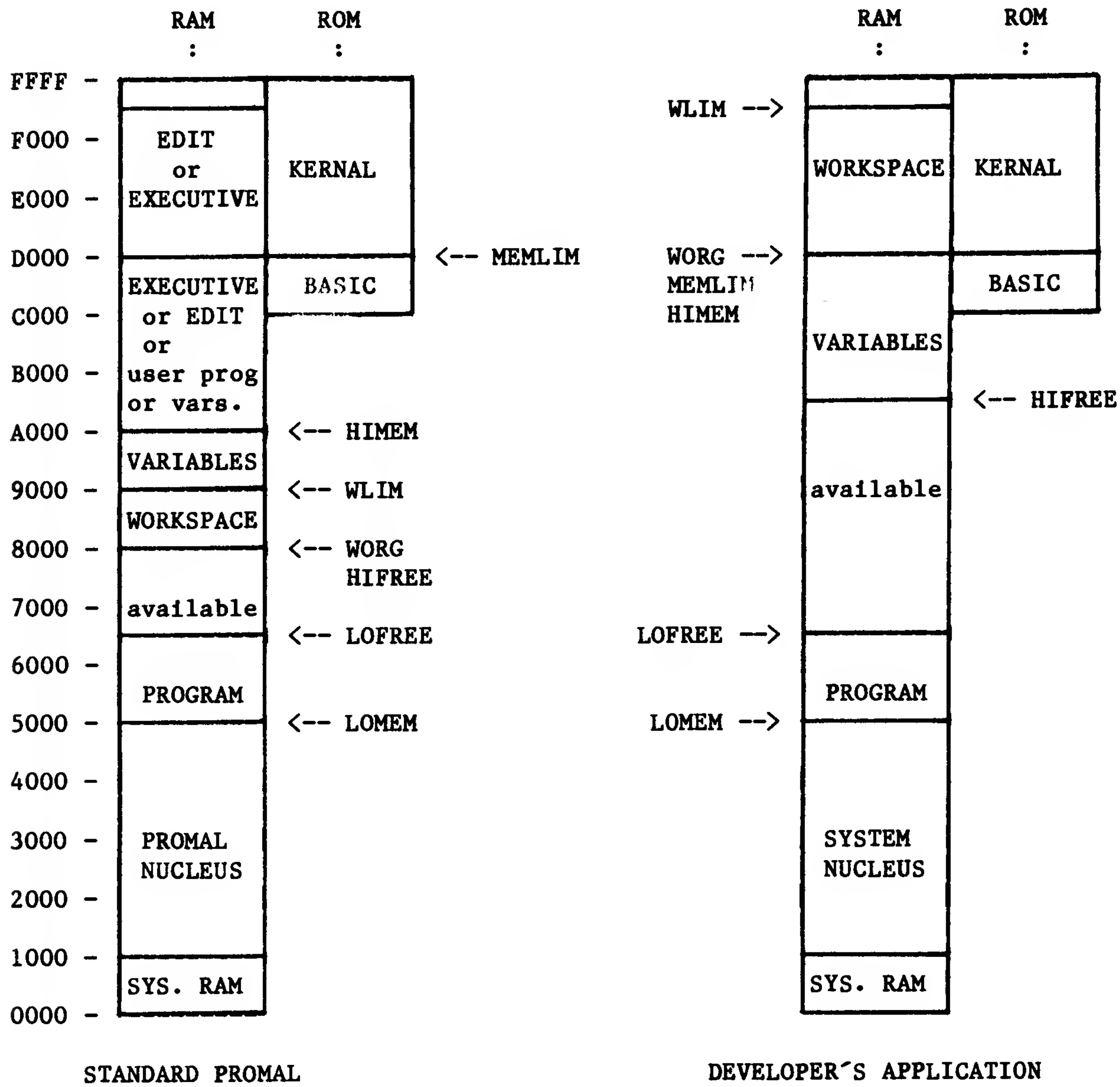
6. The EXECUTIVE automatically closes any open files when you exit a program in the normal PROMAL environment. Since you have no EXECUTIVE, you should be sure to close any files that you opened.

MEMORY MAP DIFFERENCES

Figure 1 shows the difference between the memory map for a typical compiled PROMAL program when loaded by the standard PROMAL system (left), and by the developer's master disk (right). The primary difference is that the master disk version does not have the EXECUTIVE or EDITOR in memory.

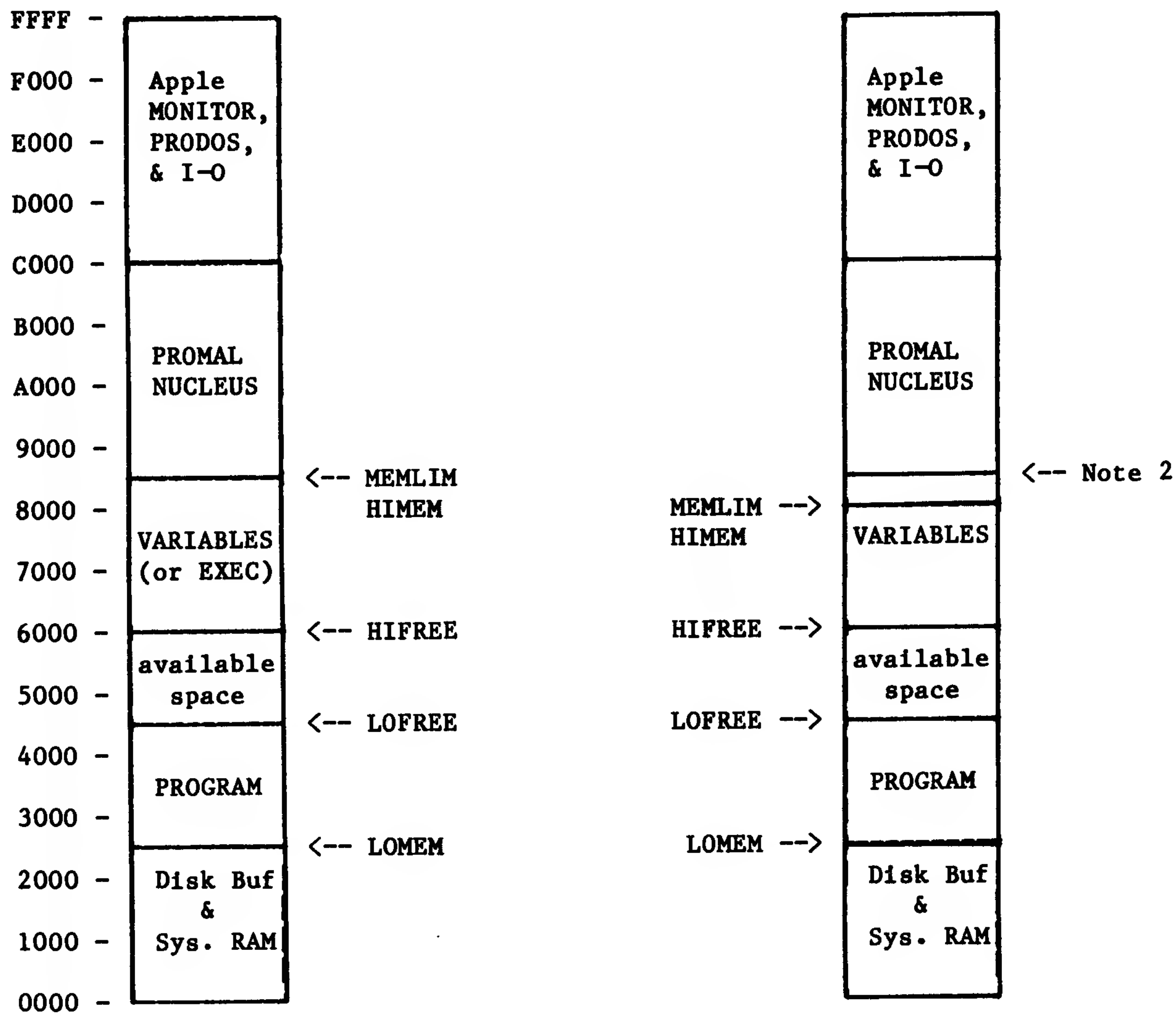
Figure 1

Memory Map - Commodore 64



Note: All addresses shown are approximate. See Appendix G of the PROMAL Language manual for locations of the memory pointers. $FE00-FFFF holds the Function key definition buffer & CTRL-B buffer (both versions).

Memory Map - Apple II



STANDARD PROMAL

DEVELOPER'S APPLICATION

Notes:

1. All addresses shown are approximate. See Appendix G of the PROMAL Language Manual for locations of the memory pointers.

2. In the Developer's version, MEMLIM is 1 page lower than in the normal version, to allow room for the Function key definitions (which are normally in banked memory). The CTRL-B buffer is in main memory too, but is included in the Nucleus for the Developer's version.

The PROMAL NUCLEUS and SYSTEM NUCLEUS are very similar, except that the SYSTEM autoloads your application program instead of the EXECUTIVE and EDITOR when it is started up (NOTE: The developer's SYSTEM file **cannot** be used to run the EXECUTIVE or EDITOR). The application PROGRAM is identical in both cases, and is simply your compiled PROMAL program.

Your application program will load in the same location as it would with the normal PROMAL system with no other programs in memory. Your variables will be loaded in the same way as normal, at the top of available memory (by default), or immediately above your program code (if you specified OWN on the PROGRAM line).

The only difference will be that on the Apple, you will have one page (256 bytes) less space at the top of free memory. This is because the Apple developer's nucleus supports the function key buffer and backtrack (CTRL-B) buffer in main memory instead of in bank-switched memory. This permits your PROMAL application to run on Apple IIe systems **without** the extra bank-switched memory. Your programs will also run on IIe computers without 80 column boards. In this case, your programs will run in 40 column mode. If you wish, your program can test for the presence of an 80 column board as described in the Apple reference manuals. Your application should not use the Workspace (W device) on Apple computers without the extra bank-switched memory present.

If your Apple II program does not use function substitution strings, you can use the page of space for a buffer, etc. To do this, move MEMLIM up one page, set the key codes for BKEYFK1 and BKEYFKL to $00 (see Appendix G), and use the page immediately below MEMLIM as you see fit.

WORKSPACE (W DEVICE) CONSIDERATIONS (COMMODORE 64 ONLY)

For the Commodore 64, another difference in the memory map is the location and size of the Workspace. When your application runs from your master diskette, the Workspace will be initialized to point to a large (11.5K byte) area starting at $D000. This will have no effect on your program unless you use the workspace from your program (the W device). This space is in the RAM of the Commodore-64 which is "under" the ROM. Reading or writing the W device automatically performs the necessary "bank-switching" to access this RAM. If you wish, you may move the location of the workspace from the application program by manipulating the pointers before starting to use the W device.

The file PROSYS.S defines five global variables associated with the Workspace:

```
WORD WORG  ($0DC5) points to the start of the available workspace.
WORD WPTR  ($0DC7) points to the next byte to be read or written.
WORD WEOF  ($0DC9) points to end-of-file (byte after the last byte written).
WORD WEND  ($0DCB) points to the byte after the last byte of the workspace.
WORD WSIZE ($0DDB) is the size of the available workspace (=WEND-WORG).
```

WORG and WEND will not change unless you change them (in the Developer's System version). This differs from the standard PROMAL system environment, where the workspace moves dynamically as programs are loaded and unloaded, and is also controlled by the WS command of the EXECUTIVE. On either system, WPTR and WEOF are moved automatically by the W device driver. A workspace with nothing in it has WEOF = WPTR = WORG.

Suppose you wish to increase the Commodore Workspace size to use all available memory. Before doing any input/output to the W device, you could execute these statements:

```
INCLUDE PROSYS
...
WORG = LOFREE   ; move start of W down to start of free memory
WPTR = WORG     ; set workspace to empty
WEOF = WORG
HIFREE = LOFREE ; no more free space now
WSIZE = WEND-WORG
...
```

Besides using the available memory for an increased workspace, you could also use it for loading machine language routines, for dynamically-allocated buffers, for hi-res screen memory, or for whatever you need. The available space starts at LOFREE and goes up to HIFREE, just as it does in the standard system; the available space is just bigger.

WORKSPACE CONSIDERATIONS (APPLE II)

The Apple II Developer's version keeps the same Workspace as the standard system.

The Workspace in the Apple is in bank-switched memory. Therefore if your application program is intended to run in IIe computers without any extra memory (40 column mode), you may not use the W device.

SIZING YOUR APPLICATION PROGRAM

For the Apple II:

Your new Master disk will have on it a copy of the ProDOS system, the PROMAL runtime system, and your compiled PROMAL application program. You should copy the PRODOS file onto the master disk after you format it using the ProDOS Utility disk. The remaining files will be copied onto the master disk when you run the GENMASTER program.

Your application program will have virtually the same amount of space and will run at the same location as it does under the standard PROMAL system. When GENMASTER runs, it will ask you if you wish to include support for REAL arithmetic in the runtime package. If you say no, this is equivalent to executing a NOREAL command from the EXECUTIVE, and will reduce the size of the runtime package by about 2.5K bytes and increase the available space for your application by the same amount. You will also be asked if you wish to reserve space for the hi-res graphics page. Answering yes is equivalent to executing a BUFFERS HIRES command from the EXECUTIVE. Another prompt will allow you to specify the number of disk buffers, which is equivalent to a BUFFERS command in the EXECUTIVE.

The amount of disk space needed will be the sum of the PRODOS file (about 15K bytes), the RUNTIME.SYSTEM file (about 19K bytes), and your compiled PROMAL application program. You may add any additional files you need to the master disk after running GENMASTER.

For the Commodore 64:

The GENMASTER program on the DEVELOPER'S DISK will copy the special runtime package plus your compiled application program onto your new Master diskette. Actually there are two runtime packages provided, one with and one without REAL support. The GENMASTER program will ask you which you wish to use. The standard SYSTEM file is approximately 18K bytes, about the same as the PROMAL file on the standard PROMAL system. Therefore your program will be loaded into memory at the same address as the start of allocatable memory in the standard version (as shown by the MAP command).

If you select the SYSTEM without REAL support, the SYSTEM file will be reduced by about 2.5K, which will reduce the load time and enable your application to be loaded at the same address as in the standard system after executing a NOREAL command.

GENMASTER will unload the EDITor from memory while it copies the system nucleus to your master disk. It will be automatically reloaded from disk later if you need it. You may not use the EDITor, EXECUTIVE, or COMPILER as an application program for GENMASTER.

The amount of disk space used on your Master disk will simply be the sum of whichever SYSTEM file you select plus the size of your compiled application program. You may copy any additional files you need using the EXECUTIVE COPY command.

EXIT, ABORT, AND RUNTIME ERROR PROCESSING

In the normal PROMAL system, when your program executes to completion (or executes a call to the EXIT or ABORT library routines), control is returned to the EXECUTIVE. When you exit from your program on a Master disk generated by the Developer's System, you don't have the EXECUTIVE to go back to, so the PROQUIT procedure will be called instead. In a dedicated application program, you may not want to have any exit at all.

This brings us to the subject of error recovery. There are several different kinds of runtime errors that can occur during execution of a PROMAL program. First, there are normal I-O errors (such as attempting to open a disk which is not in the drive), which return error indications to your program so that you may take whatever corrective action is required under program control. These kinds of errors work precisely the same under either the standard system or on your master disk system.

However, there is another class of more serious errors, the kind that "shouldn't happen" in a debugged program. In the standard system, these kinds of errors abort back to the EXECUTIVE with an error message. For example, if you divide by zero, you will get a message that says:

```
*** RUNTIME ERROR:  0-DIVIDE
AT $47BA
*** PROGRAM ABORTED.

-->
```

When you run a program from your Master Diskette, you do not have an EXECUTIVE to abort to. Instead, you can handle these kinds of errors in two ways: (1) do nothing, in which case a default error message will be printed and the computer reset; (2) provide your own error recovery using a REFUGE 3, as described below. If you do nothing and a fatal error occurs, the system will display an error message similar to this:

```
FATAL SYSTEM ERROR $0D AT $47BA
PRESS ANY KEY TO RESET COMPUTER
```

When the user presses a key, the program will exit via PROQUIT. Again, remember that this only applies to the kinds of errors that would abort a program, which should not be present in a debugged application. The meanings of the error code displayed are given in Table 1.

Table 1
Runtime Error Codes

Error #	Meaning
1	Machine language breakpoint ($00) encountered. (Note: the address of the breakpoint and register contents can be found in these locations: address - $0C51; A - $0C53; X - $0C54; Y - $0C55)
2	PROMAL breakpoint ($00) encountered at the indicated address. Usually caused by a corrupted program.
3	(reserved)
4	Stack overflow. Generally caused by too many levels of nested subroutine calls in combination with large numbers of local variables.
5	Illegal opcode. Usually caused by a corrupted program (array out of bounds, bad pointer, block move error, etc.)
6	Divide by 0 (real, integer, MOD or real overflow).
7	Required software package not loaded (for example, floating point arithmetic without REAL support loaded).
8	Illegal number of arguments on FUNC or PROC. A Library routine was called with too many or too few arguments.
9	I-O direction error. For example, trying to open the printer for read access.
A	Illegal argument for FUNC or PROC. The argument for a library routine is out of range.

B Illegal file handle. Tried to perform I-O to a file or device that was not properly opened, or missing or defective file handle argument.

C I-O redirection error. Tried to redirect to an unopened or illegal file or device.

D CTRL-STOP key pressed (Commodore 64) or CTRL-RESET (Apple II).

SPECIFYING YOUR OWN ERROR RECOVERY

If you wish, you may provide your own error recovery to recapture control within your application program after these errors. To do so, define a REFUGE 3 in your program, and control will return to this point on a fatal error. You can determine what the error was by inspection of BYTE DRTERR AT $0DF6, which will contain the error code as listed above. CAUTION: You should not add this error recovery code to your program until your application is otherwise completely debugged, as it will interfere with normal PROMAL error recovery to the EXECUTIVE. An example of error recovery is shown in the program fragment below.

```
BYTE FATAL_ERROR          ; Set false while defining REFUGE, then TRUE
EXT BYTE DRTERR AT $0DF6 ; system runtime error #
...

; Define runtime error recovery entry point...
FATAL_ERROR = FALSE       ; Don't recover while just defining refuge
; Come here on fatal runtime error only..
REFUGE 3
IF FATAL_ERROR
  OUTPUT "#CUnexpected system error $#h",DRTERR
  PUT CR,"Attempting to close all files..."
  ... ; (close files, cleanup whatever you can here)
  PUT CR,"Restarting computer now..."
  ABORT
FATAL_ERROR = TRUE        ; Turn on error recovery now
...
```

The purpose of the FATAL_ERROR variable in the fragment above is to avoid executing the recovery code itself while you are merely trying to define its location by executing the REFUGE 3 statement. Don't forget that in order for a REFUGE to be active, it must be **executed**, not just exist somewhere in your program.

LOADING MODULES AND MACHINE LANGUAGE ROUTINES

Since you cannot use the EXECUTIVE GET command to load separately compiled modules or machine language routines, your application will have to load these programs itself (if it needs any). You should use a bootstrap loader as described in Chapter 8 of the Language Manual to perform this function. This also applies to programs in the optional Graphics Toolbox. For machine language routines, you can use either the LOADer or function MLGET, as described in Appendix I.

BUILDING YOUR MASTER DISKETTE

A special PROMAL program called GENMASTER is provided on the DEVELOPER'S DISKETTE. This is an interactive program which is used to copy the PROMAL runtime nucleus and your compiled PROMAL application program onto the disk you want to become your Master Diskette (which you should have pre-formatted). For the Apple version, you should also copy the PRODOS file onto your newly formatted master disk. You should run the GENMASTER program from the regular PROMAL system. To run the program, insert the DEVELOPER'S SYSTEM DISKETTE into the drive, and type:

For the Apple II:	For the Commodore 64:
UNLOAD	
PREFIX *	UNLOAD
GENMASTER	GENMASTER

This program is self-explanatory through its prompts. Simply pressing RETURN will select the default. RETURN can also be used as a "YES" response to prompts. You will have the opportunity to specify the name of the runtime system and the program on the new master disk. For the Apple II, the runtime system name must end in ".SYSTEM" to be acceptable to ProDOS. Once your Master disk is made, it can be booted up like any other program. For the Apple II, the program will auto-boot when the computer is powered on or CTRL-APPLE-RESET is pressed. For the Commodore 64, you will type:

```
LOAD "SYSTEM",8
RUN
```

This of course assumes you accepted the default name of "SYSTEM".

COPY PROTECTION

If you are developing a commercial program, you will want to decide whether or not you wish to employ some scheme to protect your disk from illegal copying. This is entirely up to you. PROMAL neither encourages nor discourages the use of copy protection schemes. Generally any protection scheme which you can use with BASIC or machine language can also be employed with PROMAL, if you wish. If you wish to employ copy protection, consult your mass disk duplicating service or books on the subject for available techniques. A discussion of copy protection techniques or ethics is beyond the scope of this manual.

MISCELLANEOUS

1. For the Commodore 64, DYNODISK works exactly the same as for the regular system. If you simply want DYNODISK disabled in your application, you can apply the same patch to the SYSTEM or SYSTEM_NR file as is described for file PROMAL in Appendix N.

2. For the Apple II, the /RAM device will be handled exactly as in the regular system. Therefore it will be enabled if the users has more than 128K and has installed the /RAM device. Your application can be booted from /RAM in the same way as regular PROMAL.

3. If you have purchased the optional source code to PROMAL, you **may not** include it in whole or in part in any application you sell or distribute, nor can you include the Compiler (not the Demo Compiler either).

4. If you use multiple modules (as described in Chapter 8 of the PROMAL Language Manual) and have an ESCAPE in one module to a REFUGE in a separate module, if you exit from the module with the ESCAPE via a normal END, the program will still return to the parent program of the original module.

5. For IBM PC PROMAL there is only a Developer's version (no End-User's version), because programs are run under DOS.

6. A Source code listing on disk for the Developer's Runtime package (as well as the regular runtime package and library) is available as an option from SMA. Please call for ordering information.

7. For the Apple, you may wish to have your program ignore CTRL-C instead of aborting. This can be done by setting the byte at $0DE0 to $00.

FINAL NOTES

Check your DEVELOPER'S DISKETTE to see if a file called README.T is present on the disk (using the FILES command). If so, type this file before attempting to build your first Master diskette. If present, this file contains additional information not covered in this manual. Other additional files may also be provided on the Developer's disk for informational purposes.

This page is intentionally left blank

PROMAL Problem Report Form

Program problems: Registered owners may use this form for reporting problems with PROMAL system components or with the documentation. To be considered for review and correction all programming problems must be documented with:

1. Source code of an executable stand-alone program which re-creates the problem, or;
2. A sequence of commands (from the Editor or Executive) which re-creates the problem, or;
3. A combination of the above.

If possible and appropriate, printed dumps of memory locations proving or illustrating the problem should be provided.

Documentation problems: Registered owners are hereby granted permission to photocopy any pages of the PROMAL manuals containing errors, and annotate those pages to document the errors. All such photocopied pages must be attached to this form and returned to SMA, Inc.

Disclaimer: SMA, Inc. has no obligation to address, acknowledge, or correct any problem reported, nor has any liability for consequential damages resulting from any problem (see End User Agreement). However, SMA, Inc. is committed to supporting and improving the PROMAL System as an evolutionary product, and will make reasonable efforts to insure that documented problems are fixed in future releases. SMA, Inc. has no obligation to, but may from time to time, publish fixes, documentation updates, or procedures which resolve reported problems.

Sender: Name__________________________ PROMAL Version______ Serial #___________
Address____
City__________________________ State_____________ Zip____________________
Computer________________________ Daytime Telephone (____)_______________

Problem Component: __Executive __Compiler __Editor __Library __Manual

Problem Description: (Attach additional pages, if needed, and source code):

Mail to SMA, Inc. as shown on reverse side. If one page report then fold and use this form as a self-mailer. Thank you for your report.

-----------------------------------fold here---------------------------------

From ______________________
____
____
____

Place
Stamp
Here

To: Systems Management Associates, Inc.
3325 Executive Drive, Dept. PMR
P.O. Box 20025
Raleigh, NC 27619 U.S.A.

-----------------------------------fold here---------------------------------

Systems Management Associates, Inc. (SMA), 3325 Executive Drive, Raleigh, North Carolina 27609

END USER AGREEMENT, LICENSE and REGISTRATION FORM

You should carefully read the terms and conditions of this agreement before opening and using your sealed PROMAL Diskette. By opening the sealed Diskette you indicate your acceptance of this agreement. If you do not agree with these terms and conditions you should promptly return the Diskette and Manual and all other enclosed materials in unused, undamaged condition; your money will be refunded. If you accept this agreement please complete the attached registration form, sign it, and return it to SMA. Thank you.

1. Usage agreement: You agree to use the programs and manuals distributed herein known collectively as "PROMAL" as follows:

— You may use PROMAL on each computer that you as an individual, personally own, or use in your place of work. You agree not to use PROMAL on a multi-user system, a local area network, or in a time-sharing or bulletin-board service. You agree to use PROMAL only in accordance with U. S. Copyright law, and agree to copy the program only for the purpose of your personal archival storage.

— You may use the example programs or parts thereof as subroutines or parts of programs that you create for your own personal use. If you have purchased the PROMAL Developer's System, you may incorporate example code into programs that you sell or otherwise distribute.

— You may transfer the program to another party if the other party agrees to accept the terms and conditions of this agreement. All materials, whether machine readable or printed must be transferred, with you retaining nothing.

2. License for Distribution: If you purchased the PROMAL Developer's System, SMA grants you an unlimited license to sell or otherwise distribute PROMAL "run time" application programs created by the binding of the PROMAL nucleus with your object program via the provided utility program. In exchange for this license you agree to place the following acknowledgement somewhere on the packaging, the diskette, or within the documentation of every program you sell or distribute: "(your program name) is a licensed PROMAL application program. PROMAL is a registered trademark of Systems Management Associates, Inc., Raleigh, NC"

3. Term. This agreement is effective from the time you first open the sealed PROMAL diskette until you return the original diskette and any copies to SMA, transfer all to a third party or destroy the diskette(s).

4. SMA's Rights. You acknowledge that the PROMAL system and the printed Manuals are protected by U.S. Copyright law and that violation of this law could result in legal prosecution and criminal or civil penalties.

5. Limited Warranty. The programs are provided "as is" without warranty of any kind, either expressed or implied, except that:

SMA WARRANTS THAT THE PROGRAM(S) WILL OPERATE SUBSTANTIALLY IN ACCORDANCE WITH THE DOCUMENTED FUNCTIONS AND SPECIFICATIONS.

SMA does not warrant that the functions contained in the programs will meet your requirements or that the operation of the programs will be uninterrupted or error free, although due care has been taken to insure correct operation. Should the programs prove to be defective, you (and not SMA or any dealer) may have to assume the entire cost of all necessary service, repair, or correction.

SMA WARRANTS THE DISTRIBUTION DISKETTE TO BE FREE OF PHYSICAL DEFECTS IN MATERIAL AND WORKMANSHIP UNDER NORMAL USE FOR A PERIOD OF THIRTY (30) DAYS FROM DELIVERY TO YOU AS EVIDENCED BY A COPY OF YOUR PURCHASE RECEIPT.

6. Limitation of remedies. SMA's entire liability and your exclusive remedy shall be the replacement of any diskette not meeting SMA's Limited Warranty. IN NO EVENT SHALL SMA BE RESPONSIBLE FOR ANY INDIRECT OR CONSEQUENTIAL DAMAGES, EVEN IF SMA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Some states do not allow the limitation or exclusion of liability for indirect or consequential damages so the above limitation may not apply to you.

7. General. This agreement will be governed by the laws of the State of North Carolina. If any provision herein contravenes legal authority in any jurisdiction in which the agreement is performed, that provision shall be deemed to be deleted but shall not affect the validity of any other provision of this agreement.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONTITIONS. YOU ALSO AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERCEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATION BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

REGISTRATION — Sign and Mail to SMA

I have read, understand, and accept the End User Agreement for PROMAL. Please register my copy of PROMAL, Serial #__________ I understand that as a registered user I will be entitled to purchase future versions of PROMAL at reduced cost; I may be entitled to receive fixes and other enhancements (if and when provided); and I may be offered special rates on subscriptions to the PROMAL NEWSLETTER, when available.

Signed: ______________________________ Date: __________ Telephone: __________

Name: ______________________________ Address: ______________________________

City/State/Zip ______________________________

Note: This form may be used as a self-mailer, see reverse side.

fold here

From: ______________________
____
____
____

Place
Stamp
Here

Systems Management Associates
3325 Executive Drive
P. O. Box 20025
Raleigh, NC 27619

fold here

AE-E21 AD-D21
CE-E21 CD-D21

SYSTEMS MANAGEMENT ASSOCIATES
3325 Executive Drive
P.O. Box 20025
Raleigh, North Carolina 27619
(919) 878-3600

October, 1986

Dear PROMAL Customer:

Thank you for your purchase of the upgrade to Version 2.1 of PROMAL for the Apple II or Commodore 64. We are sure you will be pleased with the many improvements made to the system and with the expanded PROMAL manual.

Summarized below are the major changes from PROMAL version 2.0 to version 2.1. This information is pertinent only to users upgrading from version 2.0. Users upgrading from an earlier version than 2.0 should probably read the new manual completely due to major changes and improvements which have been made.

IMPROVEMENTS SPECIFIC TO APPLE VERSION

1. Apple PROMAL 2.1 provides full support for /RAM disk (Applied Engineering Ramworks II etc.) in the normal development mode as well as for application programs created with the Developer´s version. When PROMAL is booted up, it automatically detects the presence of more than 128K of memory, and, if found, it does not disable the /RAM device. Users of /RAM disks should have their /RAM device set up so that the first 64K bank of memory is not used for a /RAM disk but is left available for program use. This is normally no problem since it is the default setup for RAMWORKS and compatible boards. To use PROMAL with /RAM, install your /RAM device as directed by the manufacturer (for ProDOS use), then start PROMAL. For example, to start PROMAL from BASIC, type:

```
-/PROMAL/PROMAL.SYSTEM
```

Note: the prefix must be the correct prefix for the disk you wish to boot. Technical note: PROMAL detects the presence of an expanded /RAM device by testing the ID of the device (unit) in the ProDOS page. If it is $BF, then PROMAL assumes it is a standard 128K system and disables the /RAM device. If the ID is $BE (as it is for RAMWORKS), then the /RAM device is left intact.

2. Apple PROMAL 2.1 provides a handy way to access disk drives without having to specify the volume name. For example "1:MFILE" means MYFILE.C in drive 1. 1:, 2: and 0: are supported; 0: is the /RAM device. This method can be used from the Executive and programs as well. Other examples:

```
PREFIX 2:    ; Make the current path whatever is in drive 2:
FILES 0:     ; Display files in /RAM
COPY COMPERRMSG.T 0:  ; Copy to /RAM from current prefix
```

3. The PREFIX command can also be used to select any slot and drive. See the manual for details.

 (over)

4. Apple PROMAL 2.1 has a new function, ONLINE, defined in PROSYS.S which tests if a specified drive is on-line and returns the volume name.

5. Apple PROMAL 2.1 now supports path names with embedded periods, so you should be able to access any volume or path name from PROMAL.

6. For Apple PROMAL 2.1, two of the default keys for EDLINE, INLINE, GETL (and the EXECUTIVE and EDITOR) line editing have been changed:

Jump cursor to First character of line was CTRL-[is now CTRL-F
Jump cursor to Last character of line was CTRL-] is now CTRL-L.

This change was made because CTRL-[is the same code as the ESC key, and to improve the ease of use. All line-editing key definitions can now be changed (described below).

7. The W (workspace) device capacity has been greatly increased and is now a fixed size (the MAP command will display the size).

8. The EXECUTIVE COPY command has been improved. You can now specify a destination using the 1: shorthand method. You can also copy from a different prefix to the current prefix by merely specifying the source path. For example:

```
PREFIX 1:
COPY 2:MYFILE
```

copies MYFILE from drive 2 to drive 1. The form of the COPY command with the "2" as a second argument indicating two drives is no longer supported.

9. The APPLE EDITor now supports lines up to 125 characters long by scrolling and has a window feature like the Commodore version. CTRL-V sets the window to whatever column the cursor is on, and CTRL-W restores the normal left margin. Lines which have additional text off screen in either direction show the first or last character of the line in reverse video as an indication that there's more outside the viewable 80 column window.

10. A new EXTCOPY utility program is provided for copying files with wildcards supported.

11. The End User disk has FORMAT, a new disk formatting utility. Type file README.T on the End User Disk for more information.

<u>IMPROVEMENTS SPECIFIC TO THE COMMODORE 64 VERSION</u>

1. DYNODISK operation has been modified for improved reliability on all varieties of 1541 and 1571 disk drives. Operational restrictions are the same as for version 2.0.

2. GETC and GETCF from the keyboard now blink the cursor while waiting for waiting for input. Also see GETKEY below.

3. Commodore 64 GETLF now supports up to 127 character lines (instead of 80). However, if the source file/device is the keyboard, the limit is still 80.

4. In order to make more free memory available, the C-64 DATE command is now a "transient" command instead of a built-in EXECUTIVE command. This means that in order for DATE to work, you need a copy of the new DATE.C program on your boot disk. No error occurs if it is not; the date will just not be set. Unlike other programs, the DATE program is automatically unloaded after it is executed by the EXECUTIVE.

<u>IMPROVEMENTS COMMON TO BOTH VERSIONS</u>

1. All known bugs have been fixed.

2. There is more free memory (about 2 pages) available in the new version.

3. A new GETKEY function is provided in the Library. It operates similarly to GETC but does not echo to the screen. It waits for a keystroke, flashing the cursor.

4. You can now use the EXECUTIVE COPY command to copy to the S device (screen).

5. The TYPE command can type lines of up to 127 characters instead of just 80. However, on the Commodore-64, if you TYPE K then a maximum of 80 characters is supported since the keyboard buffer is only 80. The GETLF function similarly now supports up to 127 characters. The EDITor now supports lines up to 125 characters long by scrolling.

6. The COMPILER now automatically unloads a program with the same name as that just compiled if it was in memory. This prevents frustration when you think you're executing the new version but the old version is still in memory.

7. The ABS function is now in the standard LIBRARY instead of the REALFUNCS.S file. If you get a duplicate definition of ABS, you are probably using an old version of REALFUNCS.S with it still in there.

8. A global variable called DIOERR is defined in the LIBRARY. This variable reflects disk read/write errors and can be polled after GETCF, GETBLKF, PUTF, PUTBLKF, OUTPUTF to determine if a disk error occurred. DIOERR=0 is normal, 1=Disk full, 2=read error, 3=write error. You don't <u>have</u> to check DIOERR; its up to you. On the C-64, DIOERR is supported only for GETBLKF and PUTBLKF for performance reasons. The W device also sets DIOERR=2 if you write more to it than there's room for (the excess is lost).

9. You now can control the blink rate of the cursor. A new global BYTE variable BLINKD AT $0CFF controls the blink rate. Setting BLINKD to greater than $7F will cause a solid, non-blinking cursor. Setting it to 0 will totally disable the cursor (no cursor visible).

10. INLINE and EDLINE functions now have a new optional third argument which allows you to specify a starting column (and detect the ending column) for the cursor position.

11. You can now boot PROMAL without a copy of the EDITor on disk. If you need the editor later it will load from disk automatically.

12. You can, with care, change the PROMAL line editing keys. See Appendix G.

13. The manual has been substantially improved and reorganized. You may especially wish to read the new Chapter 7 of the Language Manual.

14. Source code is now available as an option for most elements of the PROMAL 2.1 system. Contact SMA for pricing.

CAUTION

You may **not** use Version 2.0 GENMASTER, EXTDIR (Apple only), or EDIT successfully with version 2.1, and visa versa. Be careful not to mix these programs with prior versions.

We are sure you will find PROMAL 2.1 to be the best program development system available on your computer. Thank you for upgrading your system.

Systems Management Associates Inc.
3325 Executive Drive
Raleigh, NC 27609
(919)878-3600